PROBLEMS, MECHANISMS AND SOLUTIONS

CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA

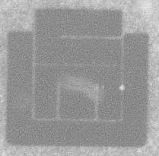
AUGUST 1976

PROBLEMS, MECHANISMS & SOLUTIONS

Ellis S. Cohen

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
August 1976

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 77 - 0006	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROBLEMS, MECHANISMS & SOLUTIONS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ellis S. Cohen		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO 2446
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE August 1976 13. NUMBER OF PAGES 161
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis formalizes the notions: problem, mechanism and solution, and shows how such a formalization is useful in describing problems and proving the correctness of solutions to them in computational systems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Mechanisms are formally defined as mappings (layers) between two computational systems. They provide natural models for protection, synchronization, and sequential and parallel control mechanisms. Certain algebraic properties of mechanisms are discussed; these correspond to properties one would ordinarily consider in studying the mechanisms listed above.

We consider those problems in computational systems that may be solved either by adding a mechanism to a system or by imposing a constraint on the states in which the system is initially permitted to operate. We find that many such problems can be described as behavioral problems, constraints on the behavior of a system. These problems may be described in a manner that is independent of the particular system. A variety of important protection problems are defined in this way.

We develop a formal methodology for solving problems in systems with multiple mechanisms. We use it in developing a number of solutions to a particular protection problem, which we solve by constraining both a protection mechanism (determining acceptable initial protection configurations) and a control mechanism (specifying properties that must be satisfied by programs which are to be executed by certain users).

Finally the thesis develops a variety of constructs for specifying behavioral problems and discusses considerations for analyzing, comparing and measuring solutions to them.

PROBLEMS, MECHANISMS & SOLUTIONS

Ellis S. Cohen

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
August 1976

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This work was supported by the Defense Advanced Research Projects Agency (#F44620-73-C-0074) where it is monitored by the Air Force Office of Scientific Research, and by the National Science Foundation under grant MCS75-07251A01.

ABSTRACT

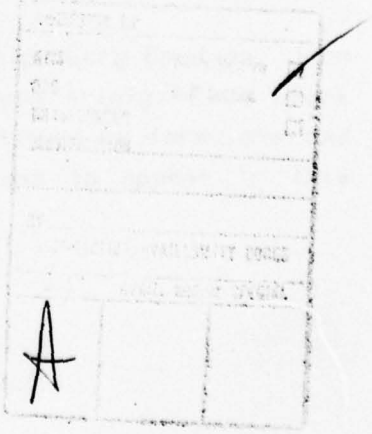
This thesis formalizes the notions: problem, mechanism and solution, and shows how such a formalization is useful in describing problems and proving the correctness of solutions to them in computational systems.

Mechanisms are formally defined as mappings (layers) between two computational systems. They provide natural models for protection, synchronization, and sequential and parallel control mechanisms. Certain algebraic properties of mechanisms are discussed; these correspond to properties one would ordinarily consider in studying the mechanisms listed above.

We consider those problems in computational systems that may be solved either by adding a mechanism to a system or by imposing a constraint on the states in which the system is initially permitted to operate. We find that many such problems can be described as behavioral problems, constraints on the behavior of a system. These problems may be described in a manner that is independent of the particular system. A variety of important protection problems are defined in this way.

We develop a formal methodology for solving problems in systems with multiple mechanisms. We use it in developing a number of solutions to a particular protection problem, which we solve by constraining both a protection mechanism (determining acceptable initial protection configurations) and a control mechanism (specifying properties that must be satisfied by programs which are to be executed by certain users).

Finally the thesis develops a variety of constructs for specifying behavioral problems and discusses considerations for analyzing, comparing and measuring solutions to them.



ACKNOWLEDGEMENTS

This thesis is dedicated to all of those whose time, love and friendship helped make this journey a joyful one. I especially want to thank the following people:

Bill Wulf, who has been my advisor since I have been at CMU. I have learned from him the value of clarity and elegance in thought and expression. Any measure of those qualities to be found in this thesis is almost wholly due to his influence. He has been supportive of the directions I've taken in pursuing this research, and has shown confidence in the end results even when my ideas were not especially coherent.

Anita Jones, chairman of the thesis committee. She has been a valuable sounding board for many of my ideas. It is impossible to estimate the extent to which her own notions about mechanisms and problems (especially relating to protection) have become incorporated in this work. She has been a careful reader of innumerable drafts of the thesis, providing additional encouragement with each one.

Nico Habermann and Garrell Pottinger, the remaining members of my committee. Nico has been helpful in pointing out ways in which my initial emphasis on protection could be smoothly broadened. His comments particularly suggested a new approach in presenting the case study. Garrell's extraordinarily careful reading of this thesis, his copious comments, and his eye for fudges and imprecisions have been responsible for many improvements in the style and notation.

John Papp, Eric Ostrom, Lee Cooprider, John Gaschnig, Gary Goodman, John Burge, Larry Flon, Larry Robinson, Jack Mostow, Gideon Ariely, Doug Clark and Dave Jefferson for many conversations that have helped me formulate and evaluate a variety of ideas, including many that came to appear in this thesis.

Dorothy Denning and Bruce Lindsay for comments on an early draft of this work.

The HYDRA group, for getting me started.

Susan Sevigny, Barb McAuley and Anita Andler, departmental secretaries, for help throughout.

The entire Computer Science department, for providing a stimulating, creative and friendly environment.

TABLE OF CONTENTS

***-may be skipped on initial reading

***-may be skipped on initial reading

***-may be skipped on initial reading

DOMAINS

$tt, ff \in TT$ - Truth Values
$\sigma \in \Sigma$ - States
$\delta \in \Delta$ - Operations
$H \in \Delta^*$ - Histories (Sequences of Operations)

$\langle \sigma, H \rangle \in BEHAVIORS = [\, \Sigma \times \Delta^* \,]$

$\varphi \in STATE\text{-}CONSTRAINT = [\, \Sigma \dashrightarrow TT \,]$
$INITIAL\text{-}CONSTRAINT = STATE\text{-}CONSTRAINT$
$\Psi \in BEHAVIORAL\text{-}CONSTRAINT = [\, BEHAVIORS \dashrightarrow TT \,]$

$\tau \in MECHANISM\text{-}MAP = [\, BEHAVIORS \dashrightarrow BEHAVIORS \,]$
usage: $\tau\langle \sigma', H' \rangle = \langle \sigma, H \rangle$
$M \in MECHANISM = [\, MECHANISM\text{-}MAP \times INITIAL\text{-}CONSTRAINT \,]$
usage: $M = \langle \tau_M, \varphi_M \rangle$

$\langle \varphi, M \rangle \in SOLUTION = INITIAL\text{-}CONSTRAINT \times MECHANISM$
$X \in PROBLEM = [\, SOLUTION \dashrightarrow TT \,]$
$STATIC\text{-}PROBLEM = STATE\text{-}CONSTRAINT$
$BEHAVIORAL\text{-}PROBLEM = BEHAVIORAL\text{-}CONSTRAINT$

INDEX OF DEFINITIONS

A NOTE ON NOTATION

Throughout this thesis, we have taken the liberty of removing universal quantifiers in logical formulae, whenever readability might be improved. For example, in definition 2-12, we write

$$\varphi_M(\sigma') \supset (\forall H')(\Psi(\tau_M\langle \sigma', H' \rangle))$$

where we should write

$$(\forall \sigma')(\varphi_M(\sigma') \supset (\forall H')(\Psi(\tau_M\langle \sigma', H' \rangle)))$$

Chapter 1 - Introduction

----- Section 1.1 --- Overview

When we interact with a complex system, be it a human system or a computing system, we inevitably find some of its behavior acceptable and some unacceptable. Just as inevitably, we tend to do what we can to bring about acceptable behavior and prevent unacceptable behavior. Such problem solving is the focus of this thesis.

If a problem is complex enough or important enough, we create mechanisms to help us solve the problem. Protection mechanisms, for example, arose from a need to solve certain protection problems, such as preventing those (unacceptable) behaviors where one user might access another user's private data.

One finds many sorts of mechanisms in computational systems. They include protection mechanisms, synchronization mechanisms, high level language mechanisms and scheduling mechanisms. These mechanisms have certain common traits. They interpose a layer between a base system and a system as provided to a user. They enhance, alter and modify the operations available in the base system, sometimes hiding base level operations, sometimes making new operations available.

In this thesis, we will present formal definitions for the terms problem, mechanism and solution. We will demonstrate how this formalism may be used in describing problems in computational systems, developing solutions for them, and proving their correctness.

This thesis focuses primarily on protection problems (also called security policies, as in [Jones & Wulf 74]). While many protection problems have been described in the literature (for example, the Mutual Suspicion Problem [Schroeder 72], Safety Problems [Harrison, Ruzzo & Ullman 75] and the Confinement Problem [Lampson 73]), no general theory of protection problems, no formal language for their specification, and no set of proof techniques has yet evolved. It is hoped that this thesis will provide a framework for such developments.

It is important that research in protection move in such a direction. The variety and quantity of information stored in computational systems continues to increase, and it is becoming increasingly more important for users to be able to exercise finer and finer control over the use of that information. Users must be able to formally specify, in as elegant a language as possible, their protection requirements. Further, they must be able to determine when and how those requirements may be met. In effect, this requires a formal theory of protection, for which the ideas in this thesis form a base.

----- Section 1.2 --- Synchronization Problems and Mechanisms

We will show that there are many parallels between synchronization and protection problems and the mechanisms designed to solve them. Formalization in synchronization has proceeded more rapidly and more generally than formalization of protection. In this section, we will survey research in synchronization, paying special attention to the development of specification and proof techniques in order to determine whether some of these might be important for, or applicable to, protection as well.

> [[Hewitt 74] contrasts synchronization and protection problems and mechanisms, in terms of a universal "actor" formalism. That work, especially the notion of behavioral specification as pursued in [Greif & Hewitt 75] and [Greif 75], has influenced many of the ideas discussed and formalized in this thesis.]

One of the earliest synchronization problems to be stated was the Mutual Exclusion Problem, discussed in [Dijkstra 68a]. When multiple users share access to an object (e.g. processor, memory, file, etc.), maintaining the integrity of the object may require that operations on it do not overlap. To prevent this overlap, a process requesting use of the object must wait until that object is not being used by any other process. Dijkstra showed how his mechanism, which introduced the operations P and V, could be used to solve the Mutual Exclusion Problem, and could in principle, be used to solve any synchronization problem at all.

However, PV proved inadequate (or at least inconvenient) for solving

certain generalizations of the Mutual Exclusion Problem. The PVmultiple mechanism [Patil 71] permitted a user to gain exclusive access to a set of objects all at once. A process making such a request would be blocked until it could be granted access to all objects requested. The PVchunk mechanism [Vantilborgh & Lamsweerde 72] was developed for those situations in which resources are represented as collections of objects (such as pages of memory or tracks of a disk), and processes request a chunk (some number) of these at a time. Using the PVchunk mechanism, a process would block until the the process could be granted exclusive access to the exact number of objects requested.

Even these mechanisms proved to be inadequate for solving more complex problems. In Reader-Writer problems [Courtois, Heymans & Parnas 71], a distinction is made between processes that read from and those that write to an object. Multiple readers may access an object simultaneously. However, writers must be given exclusive access to the object and must not overlap with any reader. Versions of the problem abound, differing in the way one assigns priorities to the readers and writers. PV mechanisms do not admit especially elegant solutions to Reader-Writer problems. Other mechanisms, subsequently introduced are better in that regard, notably the up/down mechanism [Wodon 72].

The mechanisms described above define new operations (P and V, etc.), which could be misused by users. Monitors [Hoare 74] and Regions [Hansen 73] also introduce new sets of operations but embed them in a syntactic structure to prevent misuse. (The operations for monitors are: enter monitor, wait, signal and exit monitor. Brinch Hansen's operations permit a process to block until an arbitrary condition is satisfied.) The syntactic constraints provide an important function. By imposing a structure on the solution to a synchronization problem, one might reasonably consider proving that it is correct.

Of course, a proof that a solution to a given problem is correct requires a formal specification of the problem. Initially, specifications for synchronization problems were given informally, much like our description of the problems above. The imprecision of informal specification inevitably led to controversy, in particular regarding the correctness of the PV solution to the second Reader-Writer problem as presented in [Courtois, Heymans & Parnas 71].

In order to discuss the formal specification of a problem, we must first describe the effect of a mechanism upon a system. Consider the system that happens to contain the operations, req(p,r), use(p,r) and free(p,r). These operations define operations executed by process p that request, use and free object r. The sequence of operations

req(1,r) use(1,r) free(1,r) req(2,r) use(2,r) free(2,r)

describes the behavior of the system where process 1 first requests, uses and frees r, followed by request, use and freeing of r by process 2. The sequence

req(1,r) req(2,r) use(1,r) use(2,r) free(1,r) free(2,r)

describes a situation in which process 2 uses r before r has been freed by process 1. If the system contains no synchronization mechanism, this sequence is perfectly legitimate. But if the system contains a mechanism designed to solve the Mutual Exclusion problem, this sequence is prevented from occuring. When process 2 requests access to r, it will block until r has been freed by process 1.

In general, mechanisms prevent the occurrence of certain sequences of operations. A synchronization problem can then be specified as set of acceptable sequences (for example, those in which no process uses an object that is in use by another process). A mechanism can be used to solve a synchronization problem if it permits just those sequences which are acceptable. Since we cannot expect to explicitly list those sequences which are acceptable or unacceptable, we next consider ways in which the set of acceptable sequences might be specified.

The specification techniques closest to explicitly listing the set of acceptable sequences are those which are modifications of the regular expressions of formal language theory [Hopcroft & Ullman 69]. These include event expressions [Riddle 73], path expressions [Campbell & Habermann 75], and the recent work reported in [Schneider 76].

In [Lipton 73], a problem is specified as a "solution" defined for a given mechanism. For example, Lipton defines one set of acceptable sequences to

be just those permitted by the up/down "solution" to the 2nd Reader-Writer problem as described in [Wodon 72]. Lipton then goes on to show that this set of sequences cannot be generated by using PV instead of up/down.

In [Robinson & Holt 74], a synchronization problem is specified as an invariant property of the state of the system, that is, a predicate on states that must remain satisfied over execution of any sequence of operations. The state contains pseudo-variables that are used to to encode salient characteristics of the previous behavior, such as the difference between the number of requests and frees executed.

To show that a particular mechanism can be used to solve a synchronization problem specified by an invariant, one must show that the mechanism blocks a process as long as subsequent execution of the process would result in a state not satisfying the predicate. We will find that invariants are useful for specifying protection problems as well as synchronization problems. We will refer to problems defined in this way as static problems.

[Habermann 72] first specified synchronization problems as invariants. However, he only considered those invariants which could be expressed in terms of the number of times an object has been requested or freed. Counting constructs alone are not sufficient for specifying many synchronization problems, particularly those involving priorities. [Belpaire 75] has suggested additional constructs that may be useful in developing a special purpose specification language for synchronization.

[Greif 75] argues against taking the "invariant" route. She specifies the set of acceptable sequences by imposing ordering constraints. For example (using prose in place of her notation): If operation-1 (e.g. req(1,r)) precedes operation-2 (req(2,r)) then operation-3 (free(1,r)) must precede operation-4 (use(2,r)).

The danger of invariant specifications is that by adding pseudo-variables, whose value may be altered as execution proceeds, one comes perilously close to defining specifications in terms of an implementation (the pseudo-variables becoming part of the actual state of the mechanism used to implement the specification). It should be noted that there are those,

be just those permitted by the up/down "solution" to the 2nd Reader-Writer problem as described in [Wodon 72]. Lipton then goes on to show that this set of sequences cannot be generated by using PV instead of up/down.

In [Robinson & Holt 74], a synchronization problem is specified as an invariant property of the state of the system, that is, a predicate on states that must remain satisfied over execution of any sequence of operations. The state contains pseudo-variables that are used to to encode salient characteristics of the previous behavior, such as the difference between the number of requests and frees executed.

To show that a *particular mechanism* can be used to solve a synchronization problem specified by an invariant, one must show that the mechanism blocks a process as long as subsequent execution of the process would result in a state not satisfying the predicate. We will find that invariants are useful for specifying protection problems as well as synchronization problems. We will refer to problems defined in this way as static problems.

[Habermann 72] first specified synchronization problems as invariants. However, he only considered those invariants which could be expressed in terms of the *number of times* an object has been requested or freed. Counting constructs alone are not sufficient for specifying many synchronization problems, particularly those involving priorities. [Belpaire 75] has suggested additional constructs that may be useful in developing a special purpose specification language for synchronization.

[Greif 75] argues against taking the "invariant" route. She specifies the set of acceptable sequences by imposing ordering constraints. For example (using prose in place of her notation): If operation-1 (e.g. req(1,r)) precedes operation-2 (req(2,r)) then operation-3 (free(1,r)) must precede operation-4 (use(2,r)).

The danger of invariant specifications is that by adding pseudo-variables, whose value may be altered as execution proceeds, one comes perilously close to defining specifications in terms of an implementation (the pseudo-variables becoming part of the actual state of the mechanism used to implement the specification). It should be noted that there are those,

especially [Griffiths 74], who do not find this sort of specification dangerous. Greif argues (I think correctly) that behavioral specifications (such as hers, and those based on regular expressions) have the potential for describing problems in a manner closest to our intuitive understanding of them. [This potential has not yet been fully realized, for each of the behavioral specification languages referenced treats many situations inelegantly.] Problems described in terms of behavioral specifications are termed behavioral problems, in contrast to static problems which are described in terms of invariants.

----- Section 1.3 --- Protection Problems and Mechanisms

In this section, we will survey the developments in protection problems and protection mechanisms, insofar as there are parallels with synchronization. We argue that the same formal treatment already provided for synchronization, rigorous definitions of problem, mechanism, and solution, techniques for proving the correctness of solutions, and specification languages for problems, can be usefully applied to the study of protection. All but the last are dealt with in this thesis.

We start by discussing some of the early protection mechanisms and the problems they were designed to solve. Two of these, the Access problem and the Hidden Facilities problem will be discussed more formally in sections 4.2 and 5.2 respectively.

The earliest protection mechanisms were designed to solve Access Problems. For example, the creator of (or some user reponsible for) an object (e.g. a file) may wish to prevent certain other users (or classes of users) from reading or writing the object. CTSS [Crisman 65] was one of the early systems that provided a mechanism that could be used to solve the problem. It associated an "authority list" with each file, a list of names specifying those users authorized to access the file, and the type of access permitted for each of these. The protection mechanism prevented execution of any operation that would allow an unauthorized user to access a file in a way not permitted by the authority list.

The Proprietary Program Problem can be solved using the same sort of

mechanism. A user may wish to make a proprietary program available to other users as long as they are unable to read the program (they may only execute it). The authority list for the file containing the program can contain an indication that other users are permitted to execute the file, though not to read or write it.

The introduction of the Multics protection mechanism [Organick 72, Schroeder & Saltzer 72, Saltzer 74] permitted the solution of a number of additional problems, the Hidden Facilities Problem, the Protected Subsystem Problem and the Trojan Horse Problem. We will briefly discuss each one below.

Many facilities (compilers, devices, etc.) may ordinarily be available to users of a computing utility. The Hidden Facilities Problem requires that certain users (e.g. students in programming courses) be denied access to certain of these facilities (except perhaps through the controlled use of other facilities).

The Protected Subsystem Problem is an amalgam of the Proprietary Program Problem and the Hidden Facilities Problem. A user is to be permitted to access an object, but not directly. Instead, certain programs are to made available to the user, which may be executed (but neither read nor written). Only these programs are to be able to access the object directly. The collection of programs (and the objects they maintain) are called a Protected Subsystem.

As described above, the Proprietary Program problem can be seen as characterizing a situation in which the owner of a program does not trust users of the program. The Trojan Horse Problem is concerned with the reverse situation. A user wishes to execute a borrowed program, but wants to guarantee that when executing, the program will not (maliciously or accidentally) access or damage files owned by the user that the program has no need to access. (The problem is called the Trojan Horse problem since, lurking within the depths of an elegantly crafted program made available by another user, may be a collection of code that could potentially destroy the user (or at least her files)).

It is possible that neither the owner nor the borrower of a program trust

one another. This problem, termed the Mutual Suspicion Problem [Schroeder 72], cannot be solved in Multics, though Schroeder shows how Multics may be modified to support a solution. This example illustrates a point to which we will return in discussing negotiation below. A mechanism (Multics rings) that supports solutions to some set of problems (Protected Subsystem, Trojan Horse) may not be able to support solution to combinations of these problems (Mutual Suspicion).

We have not intended to be complete in our discussion of either protection problems or protection mechanisms. Descriptions of additional problems and mechanisms used to solve them are described in [Dennis & Van Horn 66], [Lampson 71], [Gray 72], [Atwood 72], [Needham 72], [Graham & Denning 72], [Jones 73], [Cosserat 74], [Jones & Wulf 74], [Redell 74], [Redell & Fabry 74] and [Cohen & Jefferson 75]. This list does not include a class of problems we call Information Problems; we will discuss them separately in section 1.6.

While at least as many protection problems have been described in the literature as synchronization problems, no specification languages for protection have yet appeared, and not one of the protection problems discussed above has yet been specified in a suitably formal manner. However, we can characterize a protection problem in the same way that we characterized a synchronization problem - as a set of acceptable sequences of operations. For example, consider the protection problem: guarantee that Cohen cannot write the Salary file. The set of acceptable sequences are those that contain no operation whose effect is that Cohen writes the Salary file.

The reasons why an operation may not appear in a sequence are different for synchronization problems and protection problems. In synchronization, the fact that an operation (e.g. use(2,r)) cannot appear signifies that its executor must be blocked (e.g. another process is currently using the same resource). In protection problems, an operation (e.g. write(Cohen,Salary)) may not be permitted because otherwise an access violation may occur (e.g. if Cohen is not to be permitted to write the Salary file). When an operation is not permitted in a protection system, the process generally does not block. In early systems it might have been aborted; in current systems, an error may be signalled. [This does not imply that

synchronization should always imply blocking - if an executor cannot gain access to a resource, one might want to allow the user the option to use another resource instead. Similarly, when an access violation occurs, one might want to block the offending executor until it is granted the appropriate access rights.] In any case, both synchronization and protection problems can be stated in terms of acceptable sequences of operations, no matter what semantic action is intended when unacceptable behaviors are attempted.

Given that we want to formally describe a protection problem as a set of acceptable sequences, we find we are faced with the same choice as the specifier of synchronization problems. We will show in sections 4.2 and 5.2 that certain protection problems may be specified most generally (i.e. without regard to the particular protection mechanism used) as behavioral problems. Once the mechanism is specified, the problem may often be restated as a static problem, that is, as a property of the state of the entire system (including the mechanism) that is to remain invariant. While the behavioral specification is more general, it is easier to demonstrate invariance of the static specification than satisfaction of the behavioral specification.

Some protection problems have been formalized, but the specifications have all made some sort of assumption about the system in which the problem is to be solved. In [Price 73], a proof is provided of the "Chinese-American War Games Theorem" - that is, that two processes may execute concurrently yet be guaranteed to not interact. The statement of the problem is only valid though, for the particular virtual memory mechanism that Price studies.

In [Harrison, Ruzzo & Ullman 75], safety problems are discussed, those guaranteeing that no user can ever gain some particular kind of access to any object. These problems are equivalent to those of the form: Guarantee that user x cannot gain q access to object β. Of course, these problem are very strongly mechanism dependent. They are only relevant for those protection mechanisms that can be described using Lampson's access matrix (Appendix A).

[The access matrix describes at any given instant who may access

what in which way. (The rows are users, the columns are objects, and each entry lists the ways in which a user may access an object). When a user attempts to execute an operation, the mechanism prevents execution of the operation if it would cause an access to be made that is not permitted by the appropriate entry in the matrix. Access matrix systems also provide operations that permit access to be shared with or revoked from another user; these operations change the configuration of the matrix.]

In [Jones 73], a number of protection problems are specified and solutions to them are proven correct. This thesis extends that work by providing a formal notation that permits a formal specification of problems independently of the mechanisms (in Jones' case, an access-rights mechanism) used to solve them. We also develop a number of (straightforward) proof techniques that permit a rigourous (and potentially mechanizable) demonstration of the correctness of a solution.

As the use of protection in systems becomes more varied and more widespread, we will come to understand, better than we do now, what protection problems are important. We will come to see the structure of the problems we want to solve, and from that understanding, see how to build new mechanisms to help solve them. For example, a paradigm that will likely increase in importance is the negotiation paradigm. Consider a user who builds a protected subsystem for object r, and then makes this subsystem available to another user in such a way that access to r may subsequently be revoked at any time. The other user then expends some significant amount of time and energy predicated on her continuing ability to use r. She wishes a guarantee that her access to r will not be revoked (or at least not for a while). This pairing of problems is similar to that encountered in the discussion of Mutual Suspicion above, but may lead to more serious difficulties. The requirements of both users may actually conflict, and some sort of negotiation mechanism may be needed to determine whether or not they can ever be mutually satisfied. The HYDRA system [Cohen & Jefferson 75] [Levin 75] does include some mechanisms that permit negotiation, however these are far from general.

One might imagine that protection mechanisms of the future will permit users to specify (in some undetermined language) their access and control

requirements. Such negotiation mechanisms should be able to detect conflicts and possibly mediate disputes. [Rotenberg 73] has discussed the social implications of such mechanisms. [Peuto 74] has compared this feature of protection with the legal process in real estate law.

A negotiation mechanism for any reasonably complex system must be incomplete. There are conflicts in access requirements which a negotiation mechanism will be unable to detect, much less mediate, so we can never expect to find the ultimate mechanism. We do expect that a variety of interesting mechanisms, including those explicitly designed to permit specifications for negotiation, will continue to be developed. A formalization of protection problems will help us evaluate these mechanisms, for we can formally determine how well they can be used to solve important problems.

----- Section 1.4 --- Problems and Solutions

In this section, we discuss the relationship between behavioral and static problems and discuss two different classes of behavioral problems, enforcement problems and productive problems. We show how problems can be solved by imposing a constraint on a system and/or adding a new mechanism to it. This notion of solution gives rise to a methodology for solving problems that is especially useful for systems containing multiple mechanisms.

We have described problems as sets of acceptable sequences of operations. However, what is acceptable may depend upon the state of the system. In one state, some operation may have the effect of writing into the Salary file, while in another state the operation may have an entirely different effect. If we want to prevent the Salary file from being written, the operation would be unacceptable in the first case, acceptable in the second case.

In some states of the system, all sequences of operations may be acceptable. For example, in a access matrix system, it may be possible to find some initial configuration of the access matrix that guaranteed that Cohen could never gain write access to the Salary file, no matter what operations might subsequently be executed. In general, we might solve a problem by permitting the system to operate only in states in which all

sequences of operations are acceptable. We characterize those states by an *initial constraint*.

Since sequences may be acceptable in some states but not in others, we will find it convenient to call the pair < state, sequence > a behavior (since the initial state of a system and the sequence of operations executed in that state completely specify the behavior of the system). An acceptable behavior is one in which the sequence is acceptable for the initial state.

A behavioral problem is any problem that can characterized in terms of an acceptable set of behaviors. We have said that static problems are those that are specified as a property of the state of the system. However, even static problems can be seen as a shorthand for a corresponding behavioral problem. An acceptable behavior is one whose execution results in a state satisfying the specified property. The guarantee that the property remains invariant is equivalent to the guarantee that only acceptable behaviors are to be permitted.

We have assumed thus far that all behavioral problems are to be solved by guaranteeing that only acceptable behaviors are to be permitted. We will find below that productive problems are defined differently in terms of the set of acceptable behaviors; those that we have been discussing, we call Enforcement Problems. We showed above how an enforcement problem (prevent Cohen from gaining write access to the Salary file) could be solved by imposing an initial constraint on a system, and throughout this introduction we have noted how mechanisms may be added to a system to solve problems. Below, we present three examples that illustrate how these approaches interact.

-- [Impose Constraint] -- Suppose that we wished to prevent Cohen from writing the Salary file. Given a system containing an adequate protection mechanism, one might try to find an initial constraint on the states (e.g. a constraint on the initial configurations of the access matrix) that guaranteed that Cohen could never gain write access to the Salary file.

-- [Add Mechanism] -- Next, suppose that we wished to solve the same problem (preventing Cohen from writing the Salary file) in a

system containing no protection mechanism. One might choose to add the cheapest mechanism to the system that could be used to solve the problem. Instead of adding an access matrix system, one might build a mechanism that inspected each operation attempted. If execution of the operation would permit Cohen to write the Salary file, the mechanism would not permit execution of the operation.

-- [Impose Constraint & Add Mechanism] -- Finally, suppose that we were given a system that contained an incomplete access matrix mechanism, one in which Cohen might circumvent the protection mechanism by executing the operation "sneaky-write". One might add an additional mechanism to this system that simply prevented all sneaky-write's. Of course, as in the first example, to prevent Cohen from writing the Salary file through non-sneaky operations, we would still have to impose an initial constraint on the given system that prevented Cohen from gaining write access to the Salary file.

Enforcement problems are solved by guaranteeing that no unacceptable behaviors are permitted. The preceding discussion argues that this guarantee can be met either by initially constraining a system or by adding some mechanism to it. We imagine that one might like to avoid adding mechanisms when possible, and that when mechanisms must be added, they should be as simple as possible. A methodology for solving an enforcement problem might then be described as:

1. Find an initial constraint that will eliminate as many unacceptable behaviors as possible (hopefully all of them).

2. If any unacceptable behaviors remain to be prevented, add a mechanism to the system that prevents them.

In practice, we may not be given the option of adding an arbitrary mechanism to a system. Some fixed mechanism may be provided, though we imagine we can vary the behaviors prevented by the mechanism by initializing it in different ways. Our methodology for solving a problem then becomes:

1. Find an initial constraint that will eliminate as many unacceptable behaviors as possible.

2. Initialize the mechanism so that it will prevent the execution of the remaining unacceptable behaviors.

In effect, this latter approach provides a means of dealing with systems that include multiple mechanisms, where the use of one of the mechanisms is to be preferred (e.g. for reasons of simplicity or reliability). Assume that the system as given includes just the preferred mechanism, but not the other mechanisms. If an initial constraint on that given system eliminates all unacceptable behavior, then the remaining mechanisms need not be used. At worst, they will be used sparingly. We apply this approach in section 4.5 and chapter 5.

We close this section by briefly discussing Productive Problems. We found that the enforcement problem - Cohen is not to write the Salary file - might be solved (in an access matrix system) by imposing some initial constraint on the system. Yet at system initialization time, it may not be clear that Cohen's access will need to be restricted, and the requisite initial constraint may not be satisfied when the problem needs to be solved. As a result, the Salary file manager may need to solve the following problem instead: Produce a state that satisfies the requisite constraint.

That productive problem may be described in terms of acceptable behaviors - those whose execution results in a state satisfying the requisite constraint. However, it is not necessary that every behavior executed be acceptable - only that for any of a set of current states, there be at least one acceptable behavior that the Salary file manager can execute. Productive problems, like enforcement problems, can be solved by an initial constraint and a mechanism. The initial constraint defines the set of "current states" from which an acceptable sequence of operations can be executed. The mechanism can be used to represent the program that executes that acceptable sequence.

----- Section 1.5 --- Mechanisms

The description of a mechanism can be seen as analogous to that of a level of hierarchy [Dijkstra 68b], a virtual machine monitor [Popek & Goldberg 74] and decomposition of a module [Parnas 72]. Each level of an hierarchy provides a set of operations that may be used in constructing the level above. For example, a level of hierarchy corresponding to a protection mechanism might use the operation "write", supplying the operation "protected-write" to the level above.

In [Robinson 75], a level is formally specified in terms of a mapping from operations defined by the level to programs implemented in terms of operations defined by the level below. Our definition of mechanism may be considered to be a dynamic realization of such a specification. We define a mechanism as a mapping from a behavior defined at one level to a behavior defined at the level below.

While our definition of mechanism derives in large part from [Robinson 75], the idea of mapping sequences of operations to sequences of operations is adapted from [Lipton 73]. Lipton's realizations map sequences of operations provided by one synchronization mechanism to sequences of operations provided by another synchronization mechanism. Lipton's definition of a synchronization mechanism contributed to the formulation of what we call Decision Mechanisms, a class of mechanisms that can be used to model both protection and synchronization mechanisms.

Finally, the work reported in [Jones & Lipton 75] has strongly influenced our formal approach to the way in which mechanisms may be used to solve protection problems. The mechanisms defined in [Jones & Lipton 75] do not map behaviors to behaviors, but rather programs to programs. Given a program P, M is said to be a mechanism for P if, for any input, M either outputs the same value as P or a distinguished output, a violation notice. A violation notice results if execution of P on the input would violate the requirements of some given protection problem. A mechanism that gives violation notices at least as often as it should is said to be sound.

Soundness is related to the notion we call enforcement. Suppose we specify a constraint on the behaviors acceptable in a system - those

behaviors satisfying the constraint define the "acceptable" set of behaviors. If a mechanism prevents the occurrence of all of the unacceptable behaviors (analogous to sending a violation notice), the mechanism is said to enforce the constraint on behavior. Note that the mechanism may prevent acceptable behaviors from occurring as well. If the mechanism only prevents the occurrence of unacceptable behaviors, then we say that it induces rather than enforces the constraint on behavior. This is analogous to what Jones & Lipton call a maximally complete mechanism.

Jones & Lipton view a mechanism as a transformation of a program and evaluate it (e.g. soundness and completeness) on the basis of whether the result of executing the program reflects a violation of protection. We view a mechanism as constraining behavior and evaluate mechanisms more generally on the basis of the behaviors they permit.

----- Section 1.6 --- Information Problems

Information problems are those protection problems concerned with preventing the transmission of information. In this section, we briefly describe these problems and their relation to the material in this thesis.

Early research in information problems was prompted by military requirements. Each object in a military system is classified and categorized according to the information it contains. The Military Security Problem requires that no information be transmitted to a user whose clearance does not permit access to that information. The Adept-50 system [Weissman 69] was the first to include a mechanism intended to solve this problem, however, as pointed out in [Denning 76], that mechanism is flawed.

Variants of the military security problem have been described by a number of researchers and various theoretical treatments of information transmission have recently appeared, notably those by [Jones & Lipton 75] and [Denning 76]. None of these formal treatments have been based on a description of acceptable behavior, nor can they be converted to such a description in any obvious way. [Cohen 76] discusses the relationship of these approaches to a behavioral approach.

We will suggest that information problems can be specified as enforcement problems, however we show that such specifications must inherently be incomplete and incorrect, by showing that solutions to information problems do not satisfy certain properties that must hold true of solutions to enforcement problems.

In effect, we show that the initial constraint itself partially determines what behaviors are judged to be acceptable. This demonstration suggests a new formal approach to information transmission which is pursued in [Cohen 76].

----- Section 1.7 --- Plan of the Thesis

Chapter 2 introduces the basic definitions of a computational system and a mechanism. We define formally how the imposition of an initial constraint or the addition of a mechanism may enforce or induce constraints on the behavior of a system.

In chapter 3, we discuss decision mechanisms, a special class of mechanisms that can be used to model protection mechanisms, sequential and multiprogrammed control mechanisms, and synchronization mechanisms. An initial constraint of a control mechanism is shown to correspond to specification of a program or a property some program must satisfy. We also show how a system may be extended with pseudo-operations, useful for specifying a problem independently of the specification of the mechanism.

In chapter 4, we formally define behavioral and static enforcement problems and show how behavioral specifications may be converted to static ones given a system that already contains a mechanism appropriate for solving the problem. We show how these problems may be solved and why it may not be particularly important to find maximal solutions to them.

We develop a formal methodology for solving enforcement problems and illustrate its use in a system that contains both a control mechanism and a protection mechanism. We find it useful to treat the mechanisms as if they had been added in layers, with the control mechanism added to a system already presumed to include the protection mechanism.

In chapter 5, we study 3 solutions to an enforcement problem. We want to guarantee that a set of sensitive objects may be altered only by certain verified programs, in a system (described in Appendix A) containing a protection mechanism. We abstractly specify the problem as a behavioral problem (independently of the mechanism). We then show how to state the problem as a static problem, where the mechanism is assumed. The latter form is more useful for proving the correctness of the solutions. Each of the 3 solutions both constrain the protection mechanism included in the given system (restricting the initial configurations of the access matrix) and a control mechanism (dictating certain properties of verified programs) that must be added to it.

In chapter 6, productive problems are formally defined, and the methodology developed for solving enforcement problems is extended to them. We discuss in grearter detail the phenomena described in section 1.4 - the fact that many enforcement problems require the solution of a corresponding productive problem as well. Finally we consider a formal characterization of the requirement that some user be able to produce an acceptable behavior in the face of interference from other users.

In chapter 7, we formally define a problem as a characteristic function of its solutions. Such a notation is shown to be useful for evaluating, comparing, and specifying properties of solutions. We show that solutions to enforcement problems satisfy two important properties, a Containment Property, and a Join Property, neither of which are satisfied by information problems.

Chapter 8 summarizes the results of the thesis and indicates directions for future research as well as work already in progress.

Chapter 2 - Modelling Computational Systems

"We have a definite system, we name its parts, and we adopt, in many cases, a single symbol to represent each name. In doing this, forms of expression are called inevitably out of the need for them, and the proofs of theorems, which are first seen to be little more than a relatively informal direction of attention to the complete range of possibilities, become more and more recognizably indirect and formal as we proceed from our original conception...

"The discipline of mathematics is seen to be a way, powerful in comparison with others, of revealing our internal knowledge of the structure of the world, and only by the way associated with our common ability to reason and compute."

G. Spencer Brown "Laws of Form"

----- Section 2.1 --- Introduction

In this chapter we formally define a computational system in terms of states and operations. A state is comprised of uniquely named objects; operations alter the state by changing the contents of these objects when executed. Using this model, we formally define the terms executor, history and behavior.

A mechanism may be thought of as a layer interposed between a given base system and a system as presented to a user, an augmented system. The mechanism transforms operations executed by the user in the augmented system to operations executed in the base system. A part of the formal specification of a mechanism describes the mapping from behaviors in the augmented system to behaviors in the base system. In order to perform this mapping, the mechanism may require its own mechanism state, distinct from the state of the base system. The mechanism state may need to be initialized to perform properly. A specification of the required initialization completes the formal specification of a mechanism.

One might imagine cases where, in adding a mechanism, the happy hacker

might be tempted to change (albeit in small ways) the definition of the operations of the underlying base system. We will not consider such mechanisms in this thesis beyond showing that they correspond to a class of mechanisms that are not homomorphic. Homomorphism is but one of a number of algebraic properties of mechanisms that we will discuss.

The behavior of a system may be constrained by imposing a constraint on the initial state of the system or by adding a mechanism to it. We conclude the chapter by describing how mechanisms and initial constraints can be used to induce and enforce constraints on the behavior of a system.

----- Section 2.2 --- Computational Systems

--------- 2.2.a --- Discrete Serial Free Systems

A Computational System is a discrete serial free system. Such a system is a pair $\langle \Sigma, \Delta \rangle$ where Σ is the set of states of the system and Δ is the set of operations, each of which effects a state transition when executed. A system is discrete in the sense that the state does not change continually; execution of an operation causes a discrete state transition.

A system is serial in that there is no concurrent execution of operations. This does not preclude use of this model in studying systems with concurrency. It is only necessary to assume that when operations execute concurrently, the result is the same no matter how their execution is interleaved (see section 2.4).

A system is free in that the choice of the operation which is to be executed next is always completely arbitrary. No a-priori rules, either deterministic or probabilistic, determine the selection or exclusion of any operation. Such constraints are introduced separately.

--------- 2.2.b --- Objects

We will assume that the state of a computational system is comprised of a set of objects. The objects are presumed not to overlap and the contents of the objects in a given state completely define the state. In the domain of programming languages, we might associate an object with each variable. In the domain of operating systems, objects might represent files, directories, users and processes.

We assume all objects have fixed unique names taken from some set NM which may be different for each system. We will assume that NM is ordered by some lexicographic ordering. Formally, a computational system is a triple $<\Sigma,\Delta,NM>$, however in general we will not write NM explicitly and will continue to describe a computational system simply as $<\Sigma,\Delta>$.

If α is the name of an object, we write $\sigma.\alpha$ to mean the value of α in state σ. Formally we may think of a state as a vector of values - the contents of each object in the state, ordered lexicographically. That is, if $<n1,n2,...>$ is a vector of the names in NM in lexicographic order, then

$$\sigma \equiv <\sigma.n1, \sigma.n2, ...>$$

If A is a set of object names taken from NM, then we write $\sigma.A$ to represent just that portion of the state named by the objects in A. Formally, if $<\alpha1,\alpha2,...>$ is a vector of the names in A in lexicographic order, then

$$\sigma.A \equiv_{def} <\sigma.\alpha1, \sigma.\alpha2, ...>$$

This definition permits us to write

$$\sigma1.A = \sigma2.A \quad \text{for} \quad (\forall \alpha \in A)(\ \sigma1.\alpha = \sigma2.\alpha\ )$$

We define

>> Def 2-1) $\sigma1 \underset{A}{=} \sigma2$

$$\sigma1 \underset{A}{=} \sigma2 \equiv_{def} (\forall \alpha \notin A)(\ \sigma1.\alpha = \sigma2.\alpha\ )$$

That is, if $\sigma 1 \underset{A}{=} \sigma 2$, then states $\sigma 1$ and $\sigma 2$ may differ only in the values of the objects named by A. For the special case, where $\sigma 1$ and $\sigma 2$ may differ only in the value of a single object α, we define

>> Def 2-2] $\sigma 1 \underset{\alpha}{=} \sigma 2$

$$\sigma 1 \underset{\alpha}{=} \sigma 2 \equiv_{def} (\forall \alpha' \neq \alpha)(\sigma 1.\alpha' = \sigma 2.\alpha')$$

Objects may themselves have some internal structure (including pointers to other objects). However, such details are part of an interpretation and not part of our abstract model. As an example though, we might write $\sigma.x.k$ to mean the value of the k'th component of object x in state σ, and $\sigma.x[n]$ to indicate the value of the n'th element in the array-structured object x in state σ.

The objects of a system do not necessarily correspond only to the internal state of a machine that the system might model. Some of the objects may, for example, represent input and output devices. The model includes no requirement that objects be of a fixed size. Thus an arbitrarily long input stream could be modelled by an arbitrarily large object containing a representation of the input.

--------- 2.2.c --- Operations

We formally define an operation δ as a function from states to states. Semantically, we interpret

$$\delta(\sigma) = \sigma'$$

to mean that execution of operation δ in state σ results in state σ'. We assume that the result of δ is always defined.

We will find it useful to describe an operation δ in terms of an informal programming-like language. For example, suppose that δ were defined so that

$$(\forall x \neq \beta)(\delta(\sigma).x = \sigma.x)$$
$$\wedge. \quad \delta(\sigma).\beta = \sigma.\alpha$$

That is, let $\sigma\star$ be the state resulting from execution of δ in state σ. Then β's value in $\sigma\star$ is the same as α's value in state σ. The value of other objects are the same in both states. In effect, execution of δ copies α's value to β. We might define δ in the programming language-like notation

$$[\ \beta \leftarrow \alpha\ ]$$

and write

$$\delta:\ \beta \leftarrow \alpha$$

In place of $\delta(\sigma) = \sigma\star$ we may write

$$[\ \beta \leftarrow \alpha\ ](\ \sigma\ ) = \sigma\star$$

--- 2.2.d --- Executors and Generic Operations

In a computing system, we tend to associate an executor with each operation, an object (traditionally representing a user or process) presumed to be responsible for the execution of that operation. We assume that there is some function Executor and we interpret

$$\text{Executor}(\delta) = m$$

to mean that m is the object responsible for execution of δ.

The fact that the name of the executor is determined solely by the operation may conflict with the astute reader's intuition, for one could easily imagine the same operation executed by different processes. We model such an operation by a <u>set</u> of operations having the same <u>generic</u> name.

Suppose "op" is the generic name of an operation that could be executed by any process. We define a <u>set</u> of operations, op(p1), op(p2), etc. such that op(p) defines the operation op executed by process p. For each p, op(p) is a <u>distinct</u> member of Δ.

Generic operations are useful more generally for describing classes of

operations. For example, a generic "move" operation, move(p,β,α) might represent the operation executed by process p which moves the contents of α to β. For each p, β and α, move(p,β,α) is a distinct memeber of Δ. move(p,β,α) can be described as

move(p,β,α): $\beta \leftarrow \alpha$

where Executor(move(p,β,α)) = p

By defining commands in this way, the executor of a command can be determined strictly from the argument of the command itself, independently of the state in which the command is executed.

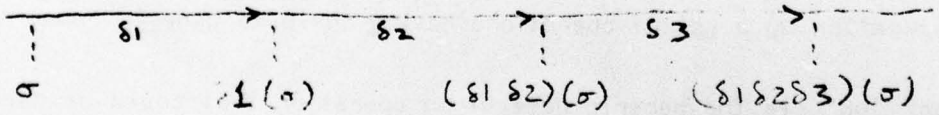
In much of the thesis, the notion of an executor may not be pertinent to the discussion. As a result, in many of the examples, we will not bother to associate an executor with an operation. In cases where we do distinguish the executor, the executor will uniformly be placed as the first "argument" of the generic name as in "move" above.

--------- 2.2.e --- Histories and Behaviors

A history is a sequence of operations. Execution of a history in a given state means sequential execution of the operations in the history. For example, if history H is the sequence of operations $\delta_1\delta_2\delta_3$, then

$$H(\sigma) = (\delta_1\delta_2\delta_3)(\sigma) = \delta_3(\delta_2(\delta_1(\sigma)))$$

Pictorially



We can define the execution of a history in a state recursively as

>> Def 2-3] H(σ) (recursively defined)

$\lambda(\sigma)$ <== σ
$(H\delta)(\sigma)$ <== $\delta(H(\sigma))$

where λ is the null history (no operations) - (not to be confused with the lambda calculus "λ" which is also used in this thesis)
and the symbol "<==" is to be read as "recursively defined as"

We write $\delta \in H$ to mean that operation δ appears in H. For example, $\delta 2 \in \delta 1 \delta 2 \delta 3$ and $\delta 4 \notin \delta 1 \delta 2 \delta 3$.

We write both HH^* as well as $H \& H^*$ to mean the concatenation of the sequences H and H^* (note "&" is not commutative).

We write $H1 \leq H2$ to mean that H1 is an initial sequence of H2. Formally

>> Def 2-4] $H1 \leq H2$

$H1 \leq H2 \equiv_{def} (\exists H)(H1 \& H = H2)$

If a system is started in state σ and some arbitrary sequence of operations H is executed, then the system exhibits some behavior which can be completely described by the pair $\langle \sigma, H \rangle$. We call a pair $\langle \sigma, H \rangle$ a behavior or a computation.

----- Section 2.3 --- Mechanisms

--------- 2.3.a --- Introduction

We may think of a mechanism as a layer interposed between a given computational system $\langle \Sigma, \Delta \rangle$ which we call the base system and the system $\langle \Sigma', \Delta' \rangle$ as provided to the user, which we call the augmented system.

When a user executes an operation δ' provided in the augmented system, the mechanism intervenes and translates δ' into execution of some sequence of operations (e.g. $\delta 1 \delta 2 \delta 3$) in the base system. The mechanism may even decide to execute no operation at all - the null history λ. If the mechanism represents a level of hierarchy, then the sequence of operations executed define the implementation of δ'.

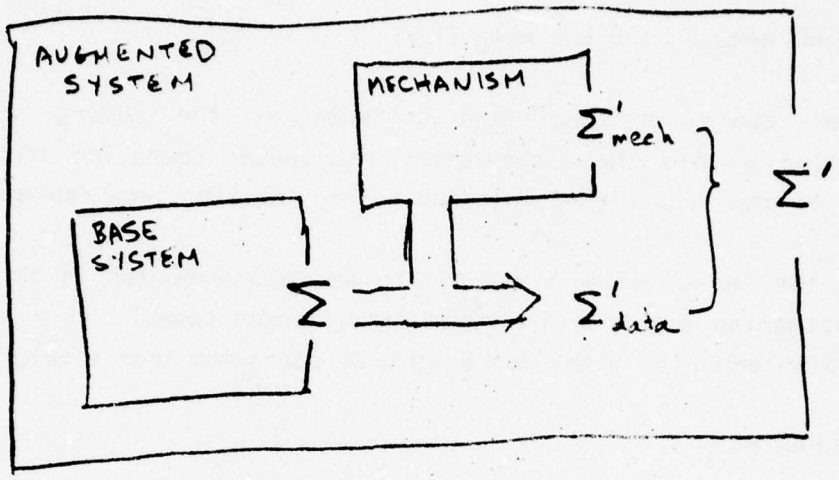
The mechanism may execute different histories for the same operation in different cases. For example, suppose a base system provided the operation

$$move(p,\beta,\alpha): \quad \beta \leftarrow \alpha$$

which, when executed by process p, copies the value of α to β. Suppose a protection mechanism augmented this system, providing the user instead with the operation $move'(p,\beta,\alpha)$. When that operation is executed, the mechanism executes $move(p,\beta,\alpha)$ in the base system only if process p has permission to read α and write β, otherwise the mechanism executes no operations at all - the null history λ.

Addition of a mechanism often (but not always - see section <gate>) implies the addition of components to the state as well - we call these additions the mechanism state. For example, when a protection mechanism is added to a system, a mechanism state component is added (in the form of additional objects, or extensions to objects already defined) which contains information about who has permission to access what. The mechanism state of a protection mechanism is often called the protection state, and is often described as an access matrix - see appendix A.

The state space Σ' of the augmented system provided to the user may be seen as consisting of two parts - the mechanism state space, which we write as Σ'_{mech} and the state space Σ of the underlying base system, which we may now think of as the data state space of the augmented system Σ'_{data}. Pictorially



The data state of the augmented system, Σ'_{data}, need not be the same as the state of the underlying base system, Σ. Instead, the mechanism may present Σ'_{data} as an abstraction of Σ. For example, if a mechanism corresponded to a module implementing an abstract data type T, then Σ'_{data} might contain objects of type T, while Σ would instead contain objects comprising the representation of elements of types used in T's representation.

While we may know that the augmented system $\langle\Sigma',\Delta'\rangle$ is composed of a base system $\langle\Sigma,\Delta\rangle$ with a mechanism M added to it, a user may be presented with $\langle\Sigma',\Delta'\rangle$ as a "black box", taking it to be a base system to which she may add a new mechanism (implement a new level).

A program language interpreter is an example of a mechanism that provides an augmented system in which operations are programming language statements and the data state consists of variables, arrays and other data structures. In the base system, operations are machine language instructions and objects include an accumulator and memory locations.

Consider the way that a programming language interpreter mechanism may map the operation

$$\delta': \ \beta \leftarrow \alpha$$

to a sequence of machine level operations. Ordinarily, δ' might be mapped to the sequence of operations in the base system.

[LOAD mem_α] [STORE mem_β]

The first operation loads the contents of the memory location mem_α representing α into the accumulator; the second operation then stores the contents of the accumulator into the memory location mem_β representing β.

As in the "move" example above, the sequence executed in the base system by the mechanism might be different in different cases. If α were known to be 0, the interpreter might map δ' into an operation that simply cleared mem_β

[CLEAR mem_β]

Finally, if α's value were known to be the the same as β's, the mechanism need map δ' into no sequence at all, that is, into the null history λ. In general, the mapping from histories executed by the user in the augmented system to histories executed by the mechanism in the base system depends upon the (data and mechanism) state.

--------- 2.3.b --- Formal Definition

Since a mechanism maps states and histories in the augmented system into states and histories in the base system, and since a state/history pair is a behavior, we can define a mechanism M in terms of of a mapping τ_M from behaviors in the augmented system to behaviors in the base system. We write

$$\tau_M\langle\sigma', H'\rangle = \langle\sigma, H\rangle$$

if the mechanism M maps state σ' to σ and history H' to H (when executed in state σ').

The mapping of the initial state is independent of the history subsequently executed in that state. That is

If $\tau_M\langle\sigma 1', H1'\rangle = \langle\sigma 1, H1\rangle$
and $\tau_M\langle\sigma 2', H2'\rangle = \langle\sigma 2, H2\rangle$

then $\sigma 1' = \sigma 2' \supset \sigma 1 = \sigma 2$

Since we can ignore the history in mapping the states, if $\tau_M\langle\sigma', H'\rangle = \langle\sigma, H\rangle$ we can make the abbreviation

$$\tau_M(\sigma') = \sigma$$

to mean that state σ' is mapped into state σ in the base system. In the example above,

$$\tau_M(\sigma').mem_\alpha = \sigma'.\alpha$$
$$\tau_M(\sigma').mem_\beta = \sigma'.\beta$$

That is, the value of α in the augmented system is the same as the value of mem_α in the base system, and similarly for β and mem_β.

We also make the abbreviation

$$\tau_M(H')(\sigma') = H$$

to mean that H' is mapped by the mechanism to H when executed in state σ'. In the example above, we would define τ_M so that

$$\tau_M(\delta')(\sigma') = \lambda \text{ if } \sigma'.\alpha = \sigma'.\beta$$
$$[\text{ CLEAR } mem_\beta\ ] \text{ if } \sigma'.\alpha = \emptyset$$
$$[\text{ LOAD } mem_\alpha\ ]\ [\text{ STORE } mem_\beta\ ] \text{ otherwise}$$

These two abbreviations are defined in such a way that

$$\tau_M\langle\sigma', H'\rangle = \langle\, \tau_M(\sigma'),\ \tau_M(H')(\sigma')\,\rangle$$

The map τ_M from states in the augmented system to states in the base system only depends upon values in the data state space. That is, if two states $\sigma 1'$ and $\sigma 2'$ have the same values in their data state part and differ only in their mechanism state part, then $\sigma 1'$ and $\sigma 2'$ must be mapped to the same state in the base system. That is

$$\tau_M(\sigma 1') = \tau_M(\sigma 2')$$

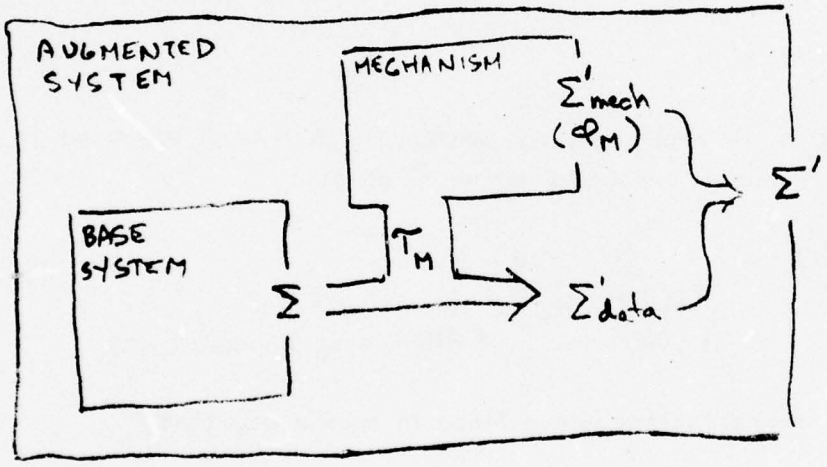
The values in the mechanism state are only used by the mechanism to

determine how histories executed by the user are to be mapped into histories executed in the base system (though the mechanism may use the data state to determine this mapping as well).

A mechanism might only work properly if its mechanism state is properly initialized. We describe this initial constraint on the mechanism state as φ_M, defined so that

$$\varphi_M(\sigma')$$

holds only if the mechanism state part of state σ' is properly initialized. We want to ensure that φ_M is defined so that it only constrains the mechanism state and not the data state. In effect we guarantee that the imposition of φ_M cannot result in the imposition of any corresponding constraint on the states of the base system.



Formally we require

$$\{ \tau_M(\sigma') \} = \{ \tau_M(\sigma') \mid \varphi_M(\sigma') \}$$

We collect the details discussed above and formally define a mechanism as follows:

>> Def 2-5] M is a mechanism from $\langle \Sigma', \Delta' \rangle$ to $\langle \Sigma, \Delta \rangle$ iff

M is a pair $\langle \tau_M, \varphi_M \rangle$ where

1. If $\tau_M\langle \sigma 1', H1' \rangle = \langle \sigma 1, H1 \rangle$
   and $\tau_M\langle \sigma 2', H2' \rangle = \langle \sigma 2, H2 \rangle$
   then $\sigma 1' = \sigma 2' \supset \sigma 1 = \sigma 2$

   [We can therefore abbreviate τ_M so that
   if $\tau_M\langle \sigma', H' \rangle = \langle \sigma, H \rangle$ we write
   $\tau_M(\sigma') = \sigma$ and $\tau_M(H')(\sigma') = H$]

2. $\{ \tau_M(\sigma') \} = \{ \tau_M(\sigma') \mid \varphi_M(\sigma') \}$

We noted above that two states $\sigma 1'$ and $\sigma 2'$ may differ only in their mechanism state (both satisfying φ_M) and therefore map to the same base state. We will find it convenient to create the notation $\sigma 1' \overset{M}{=} \sigma 2'$ for that situation. Formally

>> Def 2-6] $\sigma 1' \overset{M}{=} \sigma 2'$

$$\sigma 1' \overset{M}{=} \sigma 2' \equiv_{def} \varphi_M(\sigma 1') \wedge (\tau_M(\sigma 1') = \tau_M(\sigma 2')) \wedge \varphi_M(\sigma 2')$$

--------- 2.3.c --- Homomorphism

Earlier in this section, we provided an example of a language interpreter mechanism that would map the operation $[\beta \leftarrow \alpha]$ in the augmented system to the operation [CLEAR mem_β] in the base system if α were β. Suppose that the mechanism performed the same map in a state σ' in which $\sigma'.\alpha = 7$ instead. Now

$$[\beta \leftarrow \alpha](\sigma').\beta = 7$$

since the semantics of $[\beta \leftarrow \alpha]$ are such that (subsection 2.2.c) its execution copies the value of α to β. Since for any state σ', $\tau_M(\sigma').mem_\beta = \sigma'.\beta$, we have

$$\tau_M([\beta \leftarrow \alpha](\sigma')).mem_\beta = 7$$

However

$$[\text{CLEAR } mem_\beta](\tau_M(\sigma')).mem_\beta = 0$$

Writing $[\beta \leftarrow \alpha]$ as H', the mechanism has determined $\tau_M(H')(\sigma')$ to be $[\text{CLEAR } mem_\beta]$, yet in executing H' in the augmented system, the value of β (and therefore of mem_β) is set to 7. In effect, the mechanism has changed the semantics of a CLEAR operation so that it sets β to 7 instead of to 0. Homomorphic mechanisms correspond to a "black box" view of base systems. To add a non-homomorphic mechanism, one has to be able to "see" inside the "black box" in order to alter the definitions of the base system operations provided by it.

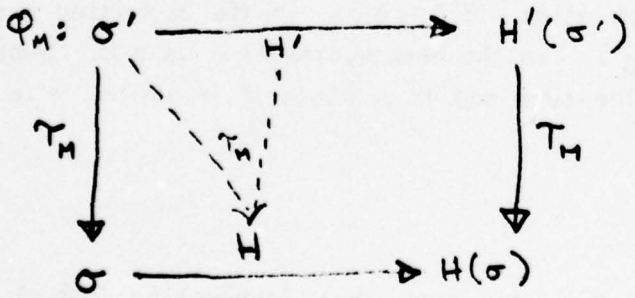
$$\tau_M(H'(\sigma')).mem_\beta = 7$$
$$(\tau_M(H')(\sigma'))(\tau_M(\sigma')).mem_\beta = 0$$

If a mechanism always correctly translates behavior in the augmented system to behavior in the base system (for any state satifying Φ_M), we say its mapping is homomorphic. Formally we define

>> Def 2-7] M is homomorphic iff

$$\Phi_M(\sigma') \supset. \quad \tau_M(H'(\sigma')) = (\tau_M(H')(\sigma'))(\tau_M(\sigma'))$$

We can describe homomorphism pictorially as



H' is executed in state σ' resulting in state H'(σ') which is mapped to σ☆

[$\tau(H'(\sigma')) = \sigma\star$]. The behavior $\langle\sigma',H'\rangle$ is mapped to $\langle\sigma,H\rangle$ [$\tau\langle\sigma',H'\rangle = \langle\sigma,H\rangle$]. If the state resulting from execution of the behavior $H(\sigma)$ is the same as $\sigma\star$, and this relation holds for any behavior $\langle\sigma,H\rangle$, then τ is homomorphic.

----- Section 2.4 --- ✲✲✲ Concurrent Mechanisms

--------- 2.4.a --- ✲✲✲ Introduction

In this section we show how a mechanism may be formalized to model concurrent execution of operations provided in the augmented system. Those mechanisms that do not model concurrency, but are sequential, are formally described as markov mechanisms.

When operations are permitted to execute concurrently, the resulting state of a computation may depend upon the way in which the operations interleave. Mechanisms in which interleaving has no effect upon the resulting state are formally defined as weakly consistent.

--------- 2.4.b --- ✲✲✲ Markov Mechanisms

Suppose that a base system provided two generic operations, $\delta 1$ and $\delta 2$, so that $\delta 1(p)$ and $\delta 2(p)$ denote the execution of $\delta 1$ and $\delta 2$ respectively by process p. That is,

$$\text{Executor}(\ \delta 1(p)\ ) = \text{Executor}(\ \delta 2(p)\ ) = p$$

Imagine that we want to guarantee that a process p can only execute the sequence of operations $\delta 1(p)\ \delta 2(p)$. We could add a mechanism to the system that would only make a single operation $\delta'(p)$ available, defined so that for all states σ'

$$\tau(\ \delta'(p)\ )(\ \sigma'\ ) = \delta 1(p)\ \delta 2(p)$$

Imagine that a process executes δ' and then executes it again. We imagine that the base system operations would be executed serially. That is

$$\tau(\ \delta'(p)\ \delta'(p)\ )(\ \sigma'\ ) \quad = \quad \delta 1(p)\ \delta 2(p)\ \delta 1(p)\ \delta 2(p)$$

However, suppose that δ' were executed by two different processes p1 and p2. In some cases, we might imagine that again, there would be no interleaving

$$\tau(\ \delta'(p1)\ \delta'(p2)\ )(\ \sigma 1'\ ) \quad = \quad \delta 1(p1)\ \delta 2(p1)\ \delta 1(p2)\ \delta 2(p2)$$

In other cases interleaving might be possible

$$\tau(\ \delta'(p1)\ \delta'(p2)\ )(\ \sigma 2'\ ) \quad = \quad \delta 1(p1)\ \delta 1(p2)\ \delta 2(p2)\ \delta 2(p1)$$

We may use the contents of the mechanism state to model variability in interleaving. The mechanism state is presumed to contain the information that tells the mechanism (e.g. - an oracle, see [Milner 72], [Cohen 75]) which process should next execute an operation in the base system. By initlalizing the mechanism state differently, the mechanism may interleave the operations differently.

We will now consider how a mechanism M might be characterized that does not permit interleaving. Suppose that when $\delta 1'$ is executed in state $\sigma 1'$, history H1 is executed in the base system; when $\delta 2'$ is executed in $\sigma 2'$, H2 is executed. If $\sigma 2'$ is the state resulting from execution of $\delta 1'$ in state $\sigma 1'$ [$\delta 1'(\sigma 1') = \sigma 2'$], then execution of $\delta 1'\delta 2'$ in state $\sigma 1'$ should <u>always</u> result in execution of H1 H2 in the base system if there is <u>no</u> interleaving. A mapping (and the mechanism it is part of) that satisfies this property for every state σ', is said to be <u>markov</u>. Formally

>> <u>Def</u> 2-8] τ is markov <u>iff</u>

$$\tau(\lambda)(\sigma') = \lambda$$
$$\tau(H'\delta')(\sigma') = \tau(H')(\sigma')\ \&\ \tau(\delta')(H'(\sigma'))$$

<u>Theorem</u> 2-1]

If τ is markov
then $\tau(H1'H2')(\sigma') = \tau(H1')(\sigma')\ \&\ \tau(H2')(H1'(\sigma'))$

>> Def 2-9] M is markov iff

τ_M is markov

If a mechanism is both markov and homomorphic, we say it is markov homomorphic.

Theorem 2-2]

If M is markov
and $\tau_M(\delta'(\sigma')) = (\tau_M(\delta')(\sigma'))(\tau_M(\sigma'))$

then M is homomorphic

--------- 2.4.c --- ☆☆☆ Weakly Consistent Mechanisms

If a mechanism is not markov and operations may be interleaved, the resulting state of a computation may differ depending upon how the operations are interleaved. For example, suppose that

$\delta 1(p)$: $\alpha \leftarrow 0$
$\delta 2(p)$: $\alpha \leftarrow \alpha + 1$

For any state σ', we find that

$(\delta 1(p1)\ \delta 2(p1)\ \delta 1(p2)\ \delta 2(p2))(\sigma').\alpha = 1$
$(\delta 1(p1)\ \delta 1(p2)\ \delta 2(p2)\ \delta 2(p1))(\sigma').\alpha = 2$

Data base system designers [Gray 75] and those interested in proving properties of parallel programs [Lipton 75] often find it important to be able to show that no matter how operations are permitted to interleave, the resulting state is the same. A mechanism that has this property is defined to be weakly consistent. Formally

>> Def 2-10] M is weakly consistent iff

$\sigma 1' \overset{M}{=} \sigma 2' \supset (\forall H')(\tau_M(H'(\sigma 1')) = \tau_M(H'(\sigma 2')))$

This characterization follows from the fact that σ1' and σ2' differ only in their mechanism state, which we noted earlier is responsible for different interleavings. The history executed in the base system if H' is executed in state σ1' is $\tau_M(H')(\sigma 1')$. The resulting state in the base system is $(\tau_M(H')(\sigma 1'))(\tau_M(\sigma 1'))$. If τ_M is homomorphic (which we will assume), the reulting state is just $\tau_M(H'(\sigma 1'))$. As a result, to determine whether interleaving affects the resulting state, we compare $\tau_M(H'(\sigma 1'))$ and $\tau_M(H'(\sigma 2'))$.

----- Section 2.5 --- Mechanisms and Behavioral Constraints

The addition of a mechanism to a base system may prevent the occurrence of certain behaviors in the base system. For example, consider the mechanism M defined so that the single operation provided, δ', is always mapped to the history δ1δ2δ3.

$$\tau_M\langle\sigma', \delta'\rangle = \langle\sigma, \delta 1 \delta 2 \delta 3\rangle$$

δ1δ2δ3 can only be executed in this sequence. The behavior $\langle\sigma, \delta 2 \delta 1 \delta 3\rangle$ cannot occur, for δ2 must follow δ1.

We can characterize those behaviors that can occur by a behavioral constraint, a predicate on behaviors, Ψ. We say that M induces Ψ if $\Psi(\sigma, H)$ is true exactly when $\langle\sigma, H\rangle$ is a behavior that can occur. For the example above, if M induces Ψ, we find that $\Psi(\sigma, \delta 1 \delta 2 \delta 3)$ but $\neg\Psi(\sigma, \delta 2 \delta 1 \delta 3)$.

Formally, the set of base system behaviors permitted by a mechanism is just

$$\{ \tau_M\langle\sigma', H'\rangle \mid \Phi_M(\sigma') \}$$

that is, the set of all behaviors which can be mapped from behaviors executed in the augmented system. We define

>> Def 2-11] M induces Ψ iff

$$\Psi(\sigma, H) \equiv \langle\sigma, H\rangle \in \{ \tau_M\langle\sigma', H'\rangle \mid \Phi_M(\sigma') \}$$

A mechanism might be added to a system in order to constrain the behavior of a system as specified by Ψ. We might call all of those behaviors that satisfy Ψ acceptable. Those that do not satisfy Ψ might be deemed unacceptable.

We might want to add a mechanism M to a system that prevents all unacceptable behaviors. Optimally, such a mechanism might only prevent unacceptable behaviors. That is, M induces Ψ. A cheaper mechanism might prevent some acceptable behaviors as well. As long as all of the unacceptable behaviors are prevented, we say that M enforces Ψ. Formally

>> Def 2-12] M enforces Ψ iff

$$\langle \sigma, H \rangle \in \{ \tau_M \langle \sigma', H' \rangle \mid \Phi_M(\sigma') \} \supset \Psi(\sigma, H)$$

or equivalently

$$(\forall \sigma', H')(\Phi_M(\sigma') \supset \Psi(\tau_M \langle \sigma', H' \rangle))$$

or alternately

$$\Phi_M(\sigma') \supset (\forall H')(\Psi(\tau_M \langle \sigma', H' \rangle))$$

In chapter 4, we will show how the set of behaviors characterized by Ψ may represent some problem we would like to solve. The addition of a mechanism may solve that problem by inducing or enforcing Ψ.

----- Section 2.6 --- Initial Constraints

--------- 2.6.a --- Constraining the Base System

We may decide to constrain a given base system so that it may not operate in certain initial states. In so doing, we prevent the occurence of certain behaviors in the system - those behaviors $\langle \sigma, H \rangle$ whose initial state σ is not one of those initially permitted. We write Φ to characterize such an initial constraint. $\Phi(\sigma)$ is true iff σ is a permitted initial state.

The behaviors induced by imposing Φ are exactly those whose initial state satisfy Φ. Formally

>> Def 2-13] Φ induces Ψ iff

$$(\forall \sigma, H)(\Phi(\sigma) \equiv \Psi(\sigma, H))$$

By preventing the occurrence of certain behaviors, the imposition of an initial constraint Φ may permit a constraint on behavior to be enforced. We define

>> Def 2-14] Φ enforces Ψ iff

$$(\forall \sigma, H)(\Phi(\sigma) \supset \Psi(\sigma, H))$$

If Φ enforces Ψ, then the imposition of Φ guarantees that only acceptable behaviors (those characterized by Ψ) will be permitted.

We introduce the notation

>> Def 2-15] $\Phi 1$ contained in $\Phi 2$

$$\Phi 1 \subseteq \Phi 2 \equiv_{def} (\forall \sigma)(\Phi 1(\sigma) \supset \Phi 2(\sigma))$$

$\Phi 1$ is contained in $\Phi 2$ if it is a stricter constraint (the set of states satisfying $\Phi 1$ are a subset of the states satisfying $\Phi 2$). By imposing stricter initial constraints, more unacceptable behaviors can be prevented.

Theorem 2-3] (proof left to reader)

If $\Phi 2$ enforces Ψ
and $\Phi 1 \subseteq \Phi 2$

then $\Phi 1$ enforces Ψ

Initial constraints may or may not be invariant. Invariance is defined as

>> Def 2-16] φ is invariant iff

$$\varphi(\sigma) \supset (\forall\delta)(\varphi(\delta(\sigma)))$$

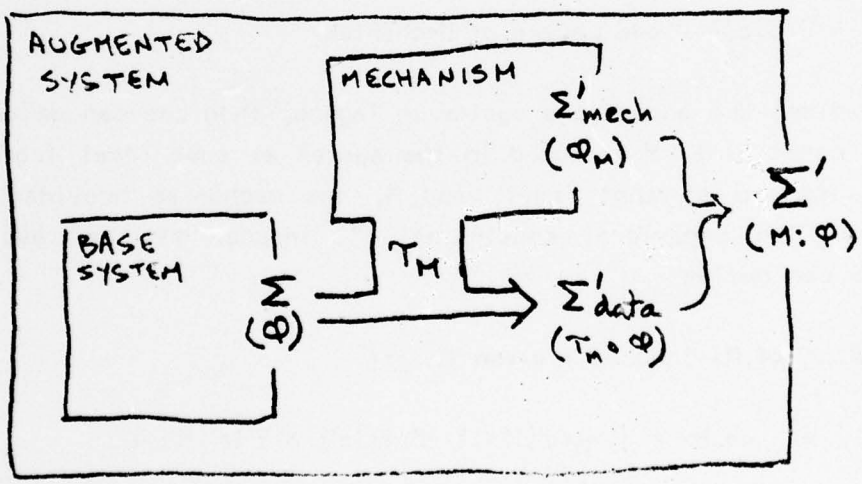
We noted in subsection 2.2.b that the contents of an object could represent a stream of input. An initial constraint could then represent restrictions on the input. In general, there is no reason to expect that φ will remain invariant as values are input from the stream.

--------- 2.6.b --- Constraining the Mechanism

We may decide to both impose an initial constraint φ on a base system as well as adding a mechanism M to it. We will find it convenient to define

>> Def 2-17] $M:\varphi$

$$(M:\varphi)(\sigma') \equiv_{def} \varphi_M(\sigma') \wedge \varphi(\tau_M(\sigma'))$$



M:φ characterizes the initial constraint on states as represented in the augmented system. It includes both the constraint on the mechanism state φ_M as well as the initial constraint φ imposed on the base system. The set of behaviors induced in the base system when φ is imposed and M is added can be characterized as:

$$\{ \tau_M\langle\sigma', H'\rangle \mid (M:\varphi)(\sigma') \}$$

We can then define

>> Def 2-18] $\langle \Phi, M \rangle$ induces Ψ iff

$$\Psi(\sigma, H) \equiv. \langle \sigma, H \rangle \in \{ \tau_M \langle \sigma', H' \rangle \mid (M:\Phi)(\sigma') \}$$

>> Def 2-19] $\langle \Phi, M \rangle$ enforces Ψ iff

$$\langle \sigma, H \rangle \in \{ \tau_M \langle \sigma', H' \rangle \mid (M:\Phi)(\sigma') \} \supset. \Psi(\sigma, H)$$

or equivalently

$$(M:\Phi)(\sigma') \supset (\forall H')(\Psi(\tau_M \langle \sigma', H' \rangle))$$

In the case that M is the identity mechanism, or Φ is the always true predicate, the reader can verify that these definitions reduce to definitions 2-13 and 2-14, and 2-11 and 2-12 respectively.

--------- 2.6.c --- *** Layers of Mechanism

If mechanisms are added to a system in layers, then one can determine the behavioral constraint Ψ induced in the system at some level from Φ, the constraint imposed at that level, and M, the mechanism provided at that level, given the behavioral constraint Ψ' induced at the level above. Formally we can define

>> Def 2-20] $\langle \Phi, M \rangle$ induces Ψ given Ψ' iff

$$\Psi(\sigma, H) \equiv. \langle \sigma, H \rangle \in \{ \tau_M \langle \sigma', H' \rangle \mid (M:\Phi)(\sigma') \wedge \Psi'(\sigma', H') \}$$

Chapter 3 - Decision Mechanisms

----- Section 3.1 --- Introduction

In this chapter we discuss a class of mechanisms, decision mechanisms, that can be used to model a wide range of protection and control (including synchronization) mechanisms. We show that decision mechanisms induce a class of behavioral constraints defined as monotonic, and further show that every monotonic behavioral constraint can be induced by some decision mechanism.

We also show how pseudo-operations may be added to a base system in order to specify synchronization problems. These pseudo-operations are meant to correspond to synchronization operations that must be provided by mechanisms added to solve those problems.

A decision mechanism is one added to a base system in order to decide, on an operation by operation basis, whether or not each operation in the base system should be permitted to execute or whether its execution should be prevented. For example, consider the base system $\langle\Sigma,\Delta\rangle$ with defined operations $\delta 1,\ldots,\delta k$. $\delta 1$ is an operation that clears the disk.

$\delta 1$: disk $\leftarrow 0$

Our task is to add a switch to this system, so that the disk may only be cleared ($\delta 1$ may only be executed) if the switch is set.

The addition of the switch augments the base system. The augmented system it produces, $\langle\Sigma',\Delta'\rangle$, has the following properties:

1. The data state space of the augmented system, Σ'_{data} is the same as the state space of the base system Σ.

2. The mechanism state space Σ'_{mech} consists of a single object "switch" whose value may be 1 (indicating that the switch is set) or 0 (indicating that the switch is not set).

3. The operations Δ' of the augmented system are $\delta 1', \ldots, \delta k'$. $\delta 2'$ through $\delta k'$ are defined exactly like $\delta 2$ through δk respectively. $\delta 1'$ may be defined as

$\delta 1'$: if switch = 1 then disk $\leftarrow$ 0

In effect, adding the switch makes the operation $\delta 1$ unavailable to the user. The augmented system instead provides $\delta 1'$ which checks the switch before performing the function of $\delta 1$, that is before clearing the disk.

The addition of the switch defines a mechanism M with the following properties:

1. $\Phi_M \equiv tt$. There is no initial constraint on the mechanism state. The initial state of the switch is not specified.

2. $\tau_M(\sigma') = \sigma'.NM$ where NM are the names of the objects in the base system (switch $\notin$ NM).

3. $\tau_M(\delta i')(\sigma') = \delta i, \quad i = 2, \ldots, k.$ The mechanism directly implements $\delta 2', \ldots, \delta k'$ as $\delta 2, \ldots, \delta k$ respectively.

4. When the user executes $\delta 1'$, the mechanism executes $\delta 1$ only if the switch is set. If the switch is not set, no operation in the base system is executed by the mechanism.

$$\tau_M(\delta 1')(\sigma') = \begin{cases} \delta 1 & \text{if } \sigma'.\text{switch} = 1 \\ \lambda & \text{if } \sigma'.\text{switch} = 0 \end{cases}$$

Having added the switch, we may feel compelled to add an operation to the system that permits the switch to be changed. We could add an operation $\delta 0'$ to the augmented system which flips the switch.

$\delta 0'$: switch $\leftarrow$ 1 - switch

The execution of $\delta 0'$ is invisible in the base system. The mechanism executes no operation in the base system when $\delta 0'$ is executed. $\delta 0'$ affects the mechanism state only. Formally, we say that $\delta 0'$ is τ_M-invisible where

>> Def 3-11 δ' is τ-invisible iff

$$(\forall \sigma')\ (\ \tau(\delta')(\sigma') = \lambda\ )$$

Protection mechanisms are excellent examples of decision mechanisms. A base system may provide operations that copy data, perform arithmetic operations and execute programs. The addition of a protection mechanism provides a new set of operations corresponding to the set provided by the base system. When an operation in the augmented system is executed by a user, the mechanism evaluates some protection condition in order to determine whether or not the corresponding operation in the base system should be executed.

The protection conditions may often be modelled as an access matrix (see Appendix A) which comprises the mechanism state. We write

$$<x,\alpha>(\sigma')$$

to describes the set of access permissions that executor x has for α in state σ'. For example,

$$r \in <x,\alpha>(\sigma') \quad \wedge \quad w \in <x,\beta>(\sigma')$$

means that the access matrix in state σ' indicates that x has the right to read ("r") α and to write ("w") β.

Where the base system might provide the operation

$$move(x,\beta,\alpha):\ \beta \leftarrow \alpha$$

the augmented system might provide the operation

$$move'(x,\beta,\alpha):\ \underline{if}\ r \in <x,\alpha> \wedge w \in <x,\beta>\ \underline{then}\ \beta \leftarrow \alpha$$

The mechanism would be defined so that

$$\tau_M(\ move'(x,\beta,\alpha)\ )(\ \sigma'\ ) =$$
$$move(x,\beta,\alpha) \quad \underline{if} \quad r \in \langle x,\alpha\rangle(\sigma') \ \wedge\ w \in \langle x,\beta\rangle(\sigma')$$
$$\lambda \quad \underline{otherwise}$$

That is, when move'(x,β,α) is executed, the mechanism executes move(x,β,α) in the base system only if the the access matrix indicates that x has the appropriate rights.

We noted above that operations invisible to the base system may be added by the mechanism in order to provide for manipulation of the mechanism state. In protection systems, these operations can be used to permit sharing and revocation of access. (Note that nothing prevents operations which are <u>not</u> invisible from altering the mechanism state as a side effect of execution as well.)

The initial constraint on the mechanism state Φ_M, represents an initial constraint on the possible configurations of the matrix.

----- <u>Section</u> 3.2 --- Formalization

Each operation provided by the augmented system (e.g. move'(x,β,α)) is mapped either a single operation in the base system (e.g. move(x,β,α)) or into no operation at all. Formally we say that the mapping τ_M is direct.

>> <u>Def</u> 3-2] τ_M is direct <u>iff</u>

$$\tau(\delta')(\sigma') \ \in\ \Delta \cup \{\lambda\}$$

where M is a mechanism from $\langle\Sigma',\Delta'\rangle$ to $\langle\Sigma,\Delta\rangle$

We want a formal definition of a decision mechanism to reflect the fact that a mechanism's decision to execute an operation in the base system is final. Subsequent behavior can have no effect on earlier decisions. Further, any decision made by the mechanism depends only upon the current state and not the previous history (except as it may be encoded in the current state. For example, the mechanism may store in the mechanism state, a record of the history executed).

If δ' has executed after H' has been executed in state σ', the determination of whether an operation should be executed in the base system or not depends only upon δ' and the current state $H'(\sigma')$. Formally, τ_M must have the property that

$$\tau_M(\lambda)(\sigma') = \lambda$$
$$\tau_M(H'\delta')(\sigma') = \tau_M(H')(\sigma') \;\&\; \tau_M(\delta')(H'(\sigma'))$$

that is, τ_M is markov (definition 2-8).

>> Def 3-3] M is a runtime mechanism iff

τ_M is direct and
τ_M is markov

Decision mechanisms are runtime mechanisms with one additional property - they are homomorphic (definition 2-7). When an operation is executed in the augmented system, a decision mechanism decides *only* whether or not a corresponding operation may be executed in the base system. Any side effects are confined to changes in the mechanism state *only*.

>> Def 3-4] M is a decision mechanism iff

M is a runtime mechanism and
M is homomorphic

The reader may note that the examples of mechanisms in the previous section were indeed homomorphic.

----- Section 3.3 --- Gatekeeper Mechanisms and Markov Constraints

In this section, we discuss gatekeeper mechanisms, those decision mechanisms which have no mechanism state. We show that such mechanisms induce a class of behavioral constraints we call markov (not to be confused with markov mappings).

If there is no mechanism state, and if the mechanism, in mapping the data

state of the augmented system to the base system does not implement an abstraction, then the state space of the augmented system and of the base system are the same, that is $\Sigma = \Sigma'$ and $\tau_M(\sigma') = \sigma'$.

Since there is no mechanism state, the mechanism can only use the current value of the data state to decide whether or not an operation should be executed in the base system. That is, to determine whether δ should be permitted when the base system state is σ, the mechanism performs some test $p(\delta)(\sigma)$. Formally, for each operation δ provided in the base system, the mechanism provides the corresponding operation δ', defined so that

$$\delta'(\sigma) = \begin{cases} \delta(\sigma) & \text{if } p(\delta)(\sigma) \\ \lambda & \text{otherwise} \end{cases}$$

This mechanism induces the behavior that can be recursively described as

$$\Psi(\sigma,\lambda) \Longleftarrow tt$$
$$\Psi(\sigma,H\delta) \Longleftarrow \Psi(\sigma,H) \wedge p(\delta)(H(\sigma))$$

That is, $\langle\sigma,H\delta\rangle$ is an acceptable behavior (satisfying Ψ) if δ is permitted in the state in which it executes - $H(\sigma)$ (i.e. if $p(\delta)(H(\sigma))$), assuming $\langle\sigma,H\rangle$ was also acceptable. We will write Ψ in such cases as

$$\text{(markov) } \Psi(\sigma,\delta) \equiv p(\delta)(\sigma)$$

We next formally define a markov behavioral constraint and show that any decision mechanism without a mechanism state induces such a constraint.

>> Def 3-5] Ψ is markov iff

$$\Psi(\sigma,\lambda) \equiv tt$$
$$\Psi(\sigma,H\delta) \equiv \Psi(\sigma,H) \wedge \Psi(H(\sigma),\delta)$$

>> Def 3-6] τ is state isomorphic iff

$$\sigma 1' = \sigma 2' \text{ iff } \tau(\sigma 1') = \tau(\sigma 2')$$

State isomorphism is a way of saying that there is no mechanism state, since any two states in the augmented system mapped into the same base state must themselves be the same. Formally, we can show that if M is a proper mechanism (definition 2-5) and τ_M is state isomorphic, then Φ_M must be the always true predicate - indicating that there is no mechanism state to constrain.

Theorem 3-1)

If M is a mechanism
and τ_M is state isomorphic

then $\Phi_M \equiv tt$

Theorem 3-2)

If M is a runtime mechanism
and τ_M is state isomorphic
and M induces Ψ

then Ψ is markov

----- Section 3.4 --- Sequential Control Mechanisms

In this section, we demonstrate how decision mechanisms may be used to model sequential control mechanisms, those that induce a base system to behave as if it were executing a sequential program.

The underlying base system defines a set Δ of operations which may be executed in any order. The mechanism state of a sequential control mechanism specifies the order of execution of the operations.

We specify a sequential control mechanism so that its mechanism state contains a specification of the program to be executed plus a pc (program counter) which indicates which operation in the program is to be executed next.

The mechanism provides a single operation δ' which can be interpreted as meaning "execute the next instruction (operation)". Suppose that the mechanism state in state σ' indicated that $\delta 4$ should be executed next. Then $\tau_M(\delta')(\sigma') = \delta 4$.

We may model a sequential control mechanism by specifying a mechanism state containing two objects. [Other models might do just as well. This one is convenient for our purposes.]

1. pc - the program counter

2. code - an object defined so that code[i] indicates the name of an operation to be executed when the value of the pc is i. code[i] either names a base system operation or a "control" operation that only alters the pc (e.g. a "goto").

For example, if

code[7] = "$\delta 4$"

then $\delta 4$ is to be executed when the value of the pc is 7. That is, if $\sigma'.pc = 7$ then $\tau_M(\delta')(\sigma') = \delta 4$. As a side effect of execution of $\delta 4$, the mechanism also adds 1 to the pc, so that $\delta'(\sigma').pc = 8$.

If the value of the pc is i and code[i] contains the name of a control operation, then $\tau_M(\delta')(\sigma') = \lambda$. A control operation has no effect on the data state, that is $\tau_M(\delta'(\sigma')) = \tau_M(\sigma')$. A control operation only changes the value of the pc, which is in the mechanism state.

We will be content to arbitrarily define three control operations: goto, branch and halt.

goto(n): pc ← n
branch(α,n): if α then pc ← n else pc ← pc + 1
halt: pc ← 0

For example, if $\sigma'.pc = 8$ and

code[8] = "goto(3)"

then $\delta'(\sigma').pc = 3$. We write

control(σ')

to mean that σ'.code[i], where i is the current value of the pc (σ'.pc), names a control operation rather than an operation defined by the base system.

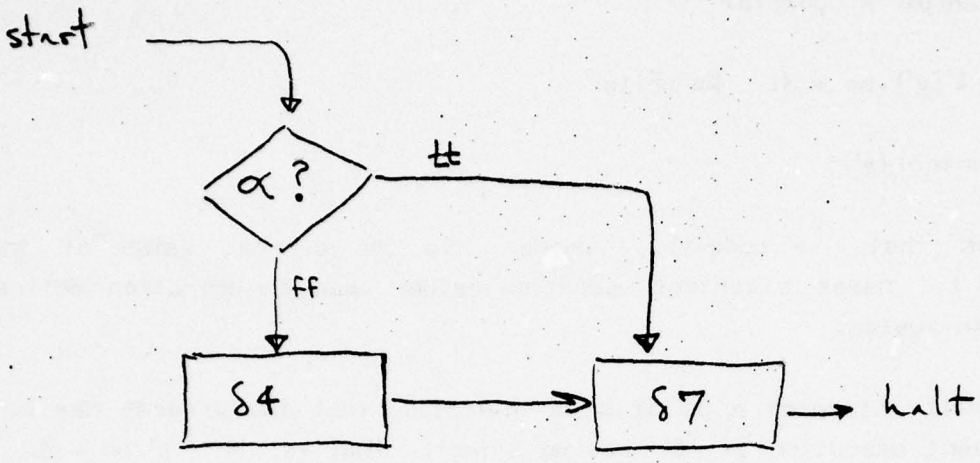
We will interpret a pc of 0 as indicating that the program has halted. Subsequent execution of δ' has no effect. That is, if $\sigma'.pc = 0$, then $\tau_M(\delta')(\sigma') = \lambda$ and $\delta'(\sigma').pc = 0$.

If NM names the objects defined in the base system (all objects but code and pc), then τ_M can formally be defined as

$$\tau_M(\sigma') = \sigma'.NM$$

$$\tau_M(\delta')(\sigma') = \begin{cases} \lambda & \underline{\text{if}} \quad \sigma'.pc = 0 \\ \lambda & \underline{\text{if}} \quad \text{control}(\sigma') \\ \delta & \underline{\text{otherwise}} \\ & \quad \underline{\text{where}} \quad \sigma'.\text{code}[\sigma'.pc] = \text{"}\delta\text{"} \end{cases}$$

Φ_M, the initial constraint on the mechanism state may specify the initial value of the pc (usually 1) and may constrain the code object as well, thereby specifying a property of the program represented in the code object. In particular, Φ_M may fully specify a program. For example, the flowchart program



can be represented by the initial constraint on the mechanism state

$\varphi_M(\sigma')$ ≡ σ'.pc = 1
∧ σ'.code[1] = "branch(α,3)"
∧ σ'.code[2] = "δ4"
∧ σ'.code[3] = "δ7"
∧ σ'.code[4] = "halt"

If $\varphi_M(\sigma')$ and $\sigma'.\alpha$ is false, then

$\tau_M(\delta')(\sigma')$ = λ (branch is a control operation)
$\tau_M(\delta'\delta')(\sigma')$ = δ4
$\tau_M(\delta'\delta'\delta')(\sigma')$ = δ4δ7
$\tau_M(\delta'\delta'\delta'\delta'....)(\sigma')$ = δ4δ7 (after δ7, the system "halts")

We can formally define δ' (the single operation supplied by the augmented system - which means "execute an operation") as

$$\delta'(\sigma').NM = \begin{cases} \sigma'.NM & \text{if } \sigma'.pc = \emptyset \\ \sigma'.NM & \text{if } control(\sigma') \\ \delta(\sigma'.NM) & \text{otherwise} \end{cases}$$
$$\text{where } \sigma'.code[\sigma'.pc] = "\delta"$$

$$\delta'(\sigma').pc = \begin{cases} \emptyset & \text{if } \sigma'.pc = \emptyset \\ \delta_{control}(\sigma').pc & \text{if } control(\sigma') \\ & \text{where } \sigma'.code[\sigma'.pc] = "\delta_{control}" \\ pc + 1 & \text{otherwise} \end{cases}$$

$$\delta'(\sigma').code = \sigma'.code$$

Note that the code component *always remains unchanged.* The reader may determine that M is indeed homomorphic, and thus M is a *decision mechanism.*

A control mechanism induces a constraint on behavior that reflects the execution of the problem it encodes. For example, the sequential control mechanism initialized according to Φ_M as defined above induces the behavioral constraint

$$\Psi(\sigma, H) \equiv H = \lambda \vee (\sigma.\alpha \wedge H = \delta 4) \vee (\sigma.\alpha \wedge H = \delta 4 \delta 7) \vee (\neg\sigma.\alpha \wedge H = \delta 7)$$

That is, if $\sigma.\alpha$ is true, the only histories permitted are λ, $\delta 4$ and $\delta 4 \delta 7$. If $\sigma.\alpha$ is false, the only histories permitted are λ and $\delta 7$.

----- Section 3.5 --- Multiprogram Control Mechanisms

In this section, we show how to extend the sequential control mechanism so that concurrent execution of a system of programs can be modelled, including synchronization operations. Instead of specifying a single pc and code object in the mechanism state, we associate a pc and code component with each executor; we call this the program component of the executor. For example, $\sigma'.x.pc$ is the value of x's pc in state σ'.

Instead of providing a single operation δ', a multiprogram control mechanism provides a set of operations $\delta'(x)$. When $\delta'(x)$ is executed in the augmented system, the mechanism executes the operation determined by x's program component.

Formally, τ_M is defined so that

$$\tau_M(\ \delta'(x)\ )(\ \sigma'\ ) = \begin{array}{ll} \lambda & \underline{\text{if}}\quad \sigma'.x.pc = 0 \\ \lambda & \underline{\text{if}}\quad control(\sigma') \\ \lambda & \underline{\text{if}}\quad Executor(\delta) \neq x \\ \delta & \underline{\text{otherwise}} \end{array}$$
$$\underline{\text{where}}\quad \sigma'.x.code[\sigma'.x.pc] = "\delta"$$

That is, when executor x next executes, the mechanism executes the operation pointed to by x's pc (unless its pc is 0) as long as the operation is supposed to be executed by executor x.

Following [Lipton 73], we show how a synchronization mechanism may be embedded in this mechanism. First, we may add objects to the mechanism state (like semaphores) that represent the synchronization state. Φ_M then also specifies the initial value of the synchronization state (the initial value of the semaphores). The operations P and V [Dijkstra 68a], when executed by x, can be modelled by the control operations

P(x,sem): if sem > 0 then
(sem ← sem - 1; x.pc ← x.pc + 1)

V(x,sem): (sem ← sem + 1; x.pc ← x.pc + 1)

Note that, if the value of sem is 0, execution of P(x,sem) leaves the value of x's pc the same, so that the next time process x executes an operation ($\delta'(x)$ is executed in the augmented system), the P is attempted again.

Suppose the code component of process i (i = 1,2) contains

code[1] = "P(i,sem)"
code[2] = "δ(i)"
code[3] = "V(i,sem)"

and the value of the pc is initially one for both processes. Then

$\tau_M(\ \delta'(1)\ )(\ \sigma'\ ) = \lambda$
[P(1,sem) executed]
$\tau_M(\ \delta'(1)\ \delta'(2)\ )(\ \sigma'\ ) = \lambda$
[P(2,sem) executes and "fails"]
$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ )(\ \sigma'\ ) = \delta(1)$
$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ )(\ \sigma'\ ) = \delta(1)$
[P(2,sem) tries again]
$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(1)\ )(\ \sigma'\ ) = \delta(1)$
[V(1,sem) executes]
$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ )(\ \sigma'\ ) = \delta(1)$
[P(2,sem) finally "succeeds"]
$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(2)\ )(\ \sigma'\ ) = \delta(1)\ \delta(2)$

[This section has described a multiprogrammed control mechanism in which a synchronization mechanism has been embedded. It is possible to define a pure synchronization mechanism that is not embedded in a control mechanism. For each operation δ provided by the base system, the mechanism would provide an operation δ' with exactly the same effect. In addition, it would provide P and V as invisible operations.]

----- Section 3.6 --- *** Mechanisms & Problem Specifications

In this section, we show how the definition of a base system may be extended to include pseudo-operations, useful in specifying the behavior of a system.

The specification of the 2nd Reader-Writer problem [Courtois, Heymans & Parnas 71] states that writers must have priority over readers. That is, if both a reader and a writer are waiting to use the same resource, then the writer will gain access before the user. It is shown in detail in [Greif 75]

that the controversy surrounded the solution is due to a certain fuzziness as to when the resource is actually requested and freed.

We show how to formally indicate when a resource has been requested or freed by adding the pseudo-operations

req(x) and free(x)

to the base system. They are both no-ops which will be executed by the mechanism when executor (process) x requests or frees a resource (we will assume a single resource here to keep things simple).

We assume that some control mechanism augments the base system. In particular, we will assume that the multiprogram control mechanism defined in the previous section is used. We will define two additional control operations that may be named in code interpreted by that mechanism.

```
Preq(x,sem):  if sem > 0
                  then  ( sem ← sem - 1;  x.pc ← x.pc + 1;
                                          x.block ← ff )
                  else  x.block ← tt

Vfree(x,sem):  sem ← sem + 1
```

These two operations are defined exactly like P and V, except that Preq sets x.block (the mechanism state is extended so that a "block" component is added to each executor) depending upon whether the process is blocked.

Most importantly, while execution of P and V is invisible in the base system, execution of Preq (its first execution only if the process becomes blocked) and Vfree and mapped into executions of req and free in the extended base system. That is, τ_M is modified so that

$$\tau_M(\ \delta'(x)\ )(\sigma') = \text{req}(x) \quad \underline{\text{if}} \quad \neg\sigma'.x.\text{block} \wedge \sigma'.x.\text{code}[\sigma'.x.\text{pc}] = \text{"Preq(x,sem)"}$$

$$\tau_M(\ \delta'(x)\ )(\sigma') = \text{free}(x) \quad \underline{\text{if}} \quad \sigma'.x.\text{code}[\sigma'.x.\text{pc}] = \text{"Vfree(x,sem)"}$$

In effect, a user of the augmented system explicitly indicates where requests and frees occur by using Preq and Vfree respectively in place of P and V. The following example, adapted from the one in the previous section, indicates how the behavior is mapped from the augmented system to the base system.

Suppose that the code component of process i (i = 1,2) contains

code[1] = "Preq(i,sem)"
code[2] = "δ(i)"
code[3] = "Vfree(i,sem)"

and the value of the pc is initially one for both processes. Then

$$\tau_M(\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(1)\ \delta'(2)\ \delta'(2)\ )(\sigma') =$$
$$\text{req}(1)\ \text{req}(2)\ \delta(1)\ \text{free}(1)\ \delta(2)$$

The set of acceptable behaviors can then be specified formally in terms of the way that ordinary base operations and the req and free operations interleave (as in [Greif 75] and [Riddle 73]).

This example suggests that problem specifications may generally require the addition of pseudo-operations to a base system that will have to correspond in some way to ordinarily invisible operations (e.g. P and V) provided by the mechanism. The pseudo-operations represent an abstract specification of the primitive actions relevant to the problem domain (req and free are relevant because we are considering synchronization problems). The definition of the mechanism indicates when these primitive actions can be considered to have taken place.

A similar approach may be useful in specifying protection problems as well, though we have not yet investigated what the appropriate primitives (pseudo-operations) might, in general, be.

----- Section 3.7 --- Monotonic Behavior

--------- 3.7.a --- Induced by Decision Mechanisms

We noted in section 2.5 that mechanisms induce constraints on the behavior observed in the base system. In this section, we will show that the class of behaviors induced by decision mechanisms can be described as monotonic.

>> Def 3-7] Ψ is monotonic iff

$$H2 \geq H1 \supset . \ \Psi(\sigma, H2) \supset \Psi(\sigma, H1)$$

Monotonicity guarantees that if some behavior $\langle\sigma, H2\rangle$ is acceptable (satisfies Ψ), then any earlier behavior $\langle\sigma, H1\rangle$, where $H1 \leq H2$, must also have been acceptable.

Theorem 3-3]

If Ψ is markov
then Ψ is monotonic

Every decision mechanism induces a monotonic behavioral constraint. More generally

Theorem 3-4]

If M is a runtime mechanism
and M induces Ψ

then Ψ is monotonic

--------- 3.7.b --- *** Construction of an Inducing Mechamnism

Any monotonic behavioral constraint can be induced by adding some decision mechanism to a base system (and possibly imposing some initial

constraint). In this section, we will show how, given an appropriate Ψ, such a mechanism, which we designate as $M_\psi = < \tau_\psi, \Phi_\psi >$, can be constructed.

For each operation in the base system, M_ψ provides a corresponding operation δ' in the augmented system. We will write $\delta' \sim \delta$ to indicate this correspondence. Whenever operation δ' is executed, the mechanism evaluates $\Psi(\sigma, H\delta)$, where $\delta' \sim \delta$, and where $<\sigma, H>$ is the behavior already executed in the base system. The mechanism executes δ only if $\Psi(\sigma, H\delta)$ is true. This implies that the mechanism must somehow remember the initial state σ and the history H already executed. The mechanism state must contain all of this information.

If NM are the names of the objects defined for (the data state) Σ, then define NM$\star$ to name a set of objects that will hold a copy of the initial state. The mechanism state consists of the objects NM$\star$ as well as an object HIST that remembers the history executed. The data state space of the augmented system is the same as the state space of the base system. That is,

$$\tau_\psi(\sigma') = \sigma'.NM$$

Initially, no history has been executed and the contents of NM and NM$\star$ are the same. Formally

$$\Phi_\psi(\sigma') \equiv \sigma'.HIST = \lambda \wedge \sigma'.NM = \sigma'.NM\star$$

For each operation δ, define δ' so that

$$\delta'(\sigma').NM\star = \sigma'.NM\star$$

$$\delta'(\sigma').NM = \underline{\text{if}}\ \Psi(\ \sigma'.NM\star,\ \sigma'.HIST\ \&\ \delta\ )\ \underline{\text{then}}\ \delta(\ \sigma'.NM\ )\ \underline{\text{else}}\ \sigma'.NM$$

$$\delta'(\sigma').HIST = \underline{\text{if}}\ \Psi(\ \sigma'.NM\star,\ \sigma'.HIST\ \&\ \delta\ )\ \underline{\text{then}}\ \sigma'.HIST\ \&\ \delta\ \underline{\text{else}}\ \sigma'.HIST$$

NM$\star$ remains the same; it remains a copy of the initial data state.

Using NM* and the past history which is in HIST, the mechanism determines whether δ can execute. If so, the data state is appropriately changed by allowing δ to execute. As a side effect, HIST is updated to reflect the fact that δ executed. Of course, τ_ψ is defined so that it is markov and

$$\tau_\psi(\delta')(\sigma') = \begin{cases} \delta & \text{if } \Psi(\sigma'.NM^*, \sigma'.HIST \,\&\, \delta), \quad \delta' \sim \delta \\ \lambda & \text{otherwise} \end{cases}$$

Theorem 3-5]

M_ψ is a decision mechanism

Theorem 3-6]

If Ψ is monotonic

then there is some Φ such that
$\langle \Phi, M_\psi \rangle$ induces Ψ

----- Section 3.8 --- *** Consistent Mechanisms

--------- 3.8.a --- *** Introduction

In previous sections, we argued that once a mechanism was added to a system, a user would know of its existence and would realize that her interaction was with the augmented system. However, one might imagine that a user does not know that a mechanism has been added, and believes she is still interacting with an original base system. The effect of each operation may appear to be the same, however, when operations are attempted, they sometimes are not executed - exactly in those cases where the decision mechanism prevents their execution.

For example, suppose a base system provides the operation move(x,β,α) which moves the contents of α to β when executed by x. A protection mechanism is added to the system and the augmented system instead provides

the operation move'(x,β,α). When this operation is executed, the mechanism only executes move(x,β,α) if x has the appropriates access rights for α and β. If x does not have the appropriate rights, the user will note that her attempted execution of move(x,β,α) fails.

We find that when decision mechanisms are added, a base system may appear to act inconsistently to an observer of the base system. We discuss weak and strong consistency - two ways in which consistency may be guaranteed. We also introduce reduction, a notation used to indicate the history actually executed given the history attempted.

--------- 3.8.b --- ☆☆☆ Strong Consistency

Mechanisms with state may appear to act capriciously or inconsistently to an observer who can only view the base system. Consider a mechanism that only permits an operation δ to execute if some switch in the mechanism state is set. An observer of the base system may find that in one case δ may initially be permitted to execute while in another case it may be prevented, even though the data (observable) state is the same in both cases. In the first case, switch is set, in the second case, it is not.

Such apparent inconsistencies cannot occur if there is no mechanism state. However, consistency may still be obtained if there is a mechanism state, as long as it is suitably constrained by φ_M. In the example above, φ_M might have initially guaranteed that switch was set, and therefore, δ would always initially be permitted.

Formally we define

>> Def 3-81 M is strongly consistent iff

$$\sigma 1' \overset{M}{=} \sigma 2' \supset (\forall H')\,(\,\tau_M(H')(\sigma 1') = \tau_M(H')(\sigma 2')\,)$$

That is, if two states are observed to be the same in the base system and the same history is executed in the augmented system in both cases, then if M is strongly consistent, the same history will be executed in the base system in both cases.

The mechanism state may be so constrained by Φ_M that only *a* single initial configuration of the mechanism state is permitted. We then say that the mechanism is strongly constrained. τ_Ψ is an example of a strongly constrained mechanism. Strongly constrained mechanisms are always strongly consistent.

>> Def 3-9] M is strongly constrained iff

$$\sigma 1' \overset{M}{=} \sigma 2' \supset . \quad \sigma 1' = \sigma 2'$$

Theorem 3-7] (proof left to reader)

If M is strongly constrained
then M is strongly consistent

Suppose that τ_M is not strongly consistent and that

$$\tau_M\langle \sigma 1', \delta' \rangle = \langle \sigma, \delta \rangle \quad \text{while} \quad \tau_M\langle \sigma 2', \delta' \rangle = \langle \sigma, \lambda \rangle$$

If M induces Ψ, then $\Psi(\sigma, \delta)$ holds because $\langle \sigma 1', \delta' \rangle$ is observed as $\langle \sigma, \delta \rangle$. However, this does not necessarily imply that execution of δ is state σ will always be allowed. In particular, both $\sigma 1'$ and $\sigma 2'$ are observed as σ (they only differ in their mechanism state), yet attempted execution of δ in the latter case will not be permitted.

In general, suppose that M induces Ψ and $\Psi(\sigma, H)$. Execution of H in state σ can only be guaranteed if M is strongly consistent.

--- 3.8.c --- ⁂ Reduction

Reduction formally describes the history actually allowed by any consistent decision mechanism (more generally, any consistent mechanism that induces a monotonic behavior) given the behavior attempted.

Suppose that the history $\delta 1 \delta 2 \delta 3$ were attempted in some base system, and the mechanism decided to permit $\delta 1$ and $\delta 3$ to execute, but not $\delta 2$. The induced behavior constraint Ψ would have the property: $\Psi(\sigma, \delta 1)$, $\neg\Psi(\sigma, \delta 1 \delta 2)$, $\Psi(\sigma, \delta 1 \delta 3)$. The history actually permitted to execute is $\delta 1 \delta 3$.

The history allowed to execute depends in general upon the state in which the history was attempted. An operation may be permitted to execute in one state but not another (even for a consistent mechanism - the decision may depend upon values in the data state). Given a consistent decision mechanism that induces Ψ, if H is attempted in state σ, we write the history actually permitted to execute as

$$H/_{\sigma}\Psi$$

Remember that the history is attempted from left to right, and the mechanism permits execution of an operation only if Ψ would be satisfied. So we can define H reduced by Ψ in σ as

>> Def 3-10] $H/_{\sigma}\Psi$ (recursively defined)

$$\lambda/_{\sigma}\Psi \;\langle==\; \lambda$$

$$(H\delta)/_{\sigma}\Psi \;\langle==$$
let R be $H/_{\sigma}\Psi$ in
if $\Psi(\sigma, R\delta)$ then $R\delta$ else R

Reduction only produces histories that satisfy Ψ.

Theorem 3-8]

If $\Psi(\sigma,\lambda)$
then $\Psi(\sigma, H/_{\sigma}\Psi)$

and if Ψ is monotonic, the set of histories that satisfy Ψ is the same as the set of reduced histories that satisfy Ψ.

Theorem 3-9]

If Ψ is monotonic
then $\Psi(\sigma,H) \supset. H = H/_{\sigma}\Psi$

It is convenient to define H reduced by Ψ as

>> Def 3-111 H/Ψ

$$H/\Psi \;=_{def}\; \lambda\sigma.\,(\,H/_{\sigma}\Psi\,)(\,\sigma\,)$$

We see that

$$(H/\Psi)(\sigma) \;=\; (\,H/_{\sigma}\Psi\,)(\,\sigma\,)$$

which is the resulting state when H is attempted in state σ.

--- 3.8.d --- ☆☆☆ Weak Consistency

When a decision mechanism that is not strongly consistent is added to a system, the state of the mechanism may affect the execution of a history attempted in the base system. However, the resulting state may be the same in any case. Formally

$$\sigma 1' \overset{M}{=} \sigma 2' \;\supset\; (\forall H')(\,\tau_M(H'(\sigma 1')) = \tau_M(H'(\sigma 2'))\,)$$

We have already defined this property (definition 2-10) as weak consistency.

Theorem 3-101

If M is strongly consistent
and M is homomorphic

then M is weakly consistent

Chapter 4 - Enforcement Problems

----- Section 4.1 --- Introduction

In this chapter we define a behavioral problem as one that can be expressed as a constraint on behavior. A class of these problems, enforcement problem, are solved by imposing an initial constraint on the system and/or adding a mechanism that enforces the behavioral constraint.

We define static problems as a special case of behavioral problems - those in which the state of a system is required to satisfy some property no matter what history is executed. A static specification of a problem is suitable when the particular system (including any mechanisms it may already contain) is specified, otherwise, a behavioral specification is generally required. When we wish to solve an enforcement problem, specified behaviorally, in a particular system, the behavioral specification can be converted to a static one. Static specifications are generally more useful for proving the correctness of solutions.

We discuss maximal solutions to problems and their relationship to the undecidability results discussed in [Harrison, Ruzzo & Ullman 75]. We indicate why maximality is not necessarily an important requirement for a solution.

Finally, we develop a methodology for determining the solution to a behavioral problem. First an initial constraint is found which eliminates some unacceptable behavior (behavior that does not satisfy the given constraint). Secondly a mechanism is found which eliminates the remianing unacceptable behavior. We present a simple example of the application of this approach in a system containing multiple mechanisms.

----- Section 4.2 --- Behavioral and Static Problems

We define a behavioral problem to be a constraint on the behavior of a system, a description of those behaviors we deem acceptable. We characterize these acceptable behaviors as $\Psi_{problem}$, so that $\Psi_{problem}(\sigma, H)$ is true only if $\langle\sigma, H\rangle$ is an acceptable behavior.

We solve a class of behavioral problems, enforcement problems, by guaranteeing that only acceptable behaviors may occur in a system. In chapter 2, we noted that unacceptable behaviors could be prevented in two ways - by imposing an initial constraint on the system, which we designate Φ_{solve} or by adding some mechanism M. That is, $\Psi_{problem}$ may be solved by finding a pair $\langle\Phi_{solve}, M\rangle$ such that

$\langle\Phi_{solve}, M\rangle$ enforces $\Psi_{problem}$

We do not require that $\langle\Phi_{solve}, M\rangle$ induce $\Psi_{problem}$. As a result, acceptable behaviors may be prevented by the solution as well as unacceptale ones. If our goal is to prevent as few unnecessary behaviors as possible, then a solution that induces $\Psi_{problem}$ would certainly be optimal. However, one might imagine other criteria for determining adequate or optimal solutions. These are discussed in section 7.3.

We may want to guarantee that some property of the state of a system remain satisfied over execution of any history. We will call such a problem a static enforcement problem and describe it as $\Phi_{problem}$ where $\Phi_{problem}(\sigma)$ is true only if σ satisfies the given property.

An example of a static enforcement problem is the following Access Problem (section 1.3): Cohen should never gain write access to the Salary file. We can represent this formally as

$$\Phi_{problem}(\sigma) \equiv w \notin \langle Cohen, Salary\rangle(\sigma)$$

Static enforcement problems include the safety problems discussed in [Harrison, Ruzzo & Ullman 75]. The safety problem for right q is the same as the static problem

$$\varphi_{problem}(\sigma) \equiv (\forall x,y)(\, q \notin \langle x,y\rangle(\sigma)\,)$$

That is, the problem is to guarantee that no object may ever have q-rights for an object.

The static problem $\varphi_{problem}$ can be seen as a shorthand notation for the behavioral problem $\Psi_{problem}$ defined so that

$$\Psi_{problem}(\sigma,H) \equiv \varphi_{problem}(H(\sigma))$$

That is, a behavior $\langle\sigma,H\rangle$ is acceptable only if its resulting state, $H(\sigma)$, satisfies $\varphi_{problem}$.

We defined (definition 2-19) $\langle\varphi_{solve},M\rangle$ enforces $\Psi_{problem}$ as

$$(M{:}\varphi_{solve})(\sigma') \supset (\forall H')(\, \Psi_{problem}(\tau_M\langle\sigma',H'\rangle)\,)$$

We can alter that definition in the following way so that it refers to static problems.

>> Def 4-1] $\langle\varphi_{solve},M\rangle$ enforces $\varphi_{problem}$ iff

$$(M{:}\varphi_{solve})(\sigma') \supset (\forall H')(\, \varphi_{problem}(\tau_M(H'(\sigma')))\,)$$

For the cases where only the initial constraint or the mechanism is used to solve $\varphi_{problem}$, we define

>> Def 4-2] φ_{solve} enforces $\varphi_{problem}$ iff

$$\varphi_{solve}(\sigma) \supset (\forall H)(\, \varphi_{problem}(H(\sigma))\,)$$

>> Def 4-3] M enforces $\varphi_{problem}$ iff

$$\varphi_M(\sigma') \supset (\forall H')(\, \varphi_{problem}(\tau_M(H'(\sigma')))\,)$$

In section 1.3, we argued that static specifications are useful given a particular system, while behavioral specifications are suited to more general problem statements. For example, the statement of the access problem

$$\Phi_{problem}(\sigma) \equiv w \notin \langle Cohen, Salary \rangle(\sigma)$$

implied an access matrix mechanism that permits Cohen to modify Salary only when Cohen has write access to it (as in the system described in appendix A).

In order to state a more general behavioral specification, we need to assume a general predicate "Wacc". We write $Wacc(\alpha, \sigma, \delta)$ to mean that in executing δ in state σ, a write access is made to object α. An appropriate statement of the problem would then be

$$(markov)\ \Psi_{problem}(\sigma, \delta) \equiv$$
$$Cohen = Executor(\delta) \supset \neg Wacc(Salary, \sigma, \delta)$$

That is, Cohen is never to be permitted to execute an instruction that causes a write access to be made to the Salary file.

When we wish to solve this problem in a system in which an appropriate mechanism is already included, we can convert the behavioral specification to a static one. In the system defined in Appendix A, if Cohen does not have write access to the Salary file, then no operation executed by Cohen can cause a write access to Salary. Formally,

$$w \notin \langle Cohen, Salary \rangle(\sigma) \supset$$
$$(\forall \delta)(Cohen = Executor(\delta) \supset \neg Wacc(Salary, \sigma, \delta))$$

As a result, we can convert the specification of $\Psi_{problem}$ to $\Phi_{problem}$. Formally, this follows from the theorem

Theorem 4-1]

If Φ_{solve} enforces $\Phi_{problem}$
and $\Psi_{problem}$ is markov
and $\Phi_{problem}(\sigma) \supset (\forall \delta)(\Psi_{problem}(\sigma, \delta))$

then Φ_{solve} enforces $\Psi_{problem}$

The value of such a conversion will be demonstrated more forcefully in chapter 5

----- Section 4.3 --- Maximal Solutions

Different combinations of initial constraints and mechanisms lead to a variety of different solutions to behavioral problems. However, even if we restrict our attention to initial constraints alone, there may be more than one solution. We may find more than one φ_{solve} that *enforces* $\Psi_{problem}$, that is, more than one φ_{solve} such that

$$\varphi_{solve}(\sigma) \supset (\forall H)(\Psi_{problem}(\sigma,H)) \qquad (\text{"}\uparrow\text{"-arrows for emphasis})$$

However, there is a maximal solution, one that least constrains the set of initial states. It is

$$\varphi_{max}(\sigma) \equiv (\forall H)(\Psi_{problem}(\sigma,H))$$

[We note here that the maximal solution φ_{max} does not necessarily induce $\Psi_{problem}$. For example, consider an arbitrary system with a single operation δ. The maximal solution to

$$\Psi_{problem}(\sigma,H) \equiv H = \lambda \vee H = \delta$$

is $\varphi_{max}(\sigma) \equiv ff$

which certainly does not induce $\Psi_{problem}$.]

The following example illustrates the value of non-maximal solutions. Consider a system that includes as two objects, a bank vault and a robber. The robber can rob the bank unless the vault is locked or the robber is drunk. For now, the only action in this system we will focus on is the attempted robbery, which can be represented by the single operation

```
δ1:  if ¬drunk(robber) ∧ ¬locked(vault)  then
         ( robber.money ← robber.money + vault.money;
           vault.money ← 0 )
```

Our problem is to find some way to guarantee that the robber can get no

money by robbing the vault. This can be stated formally by the following markov behavioral constraint

$$\text{(markov)}\ \Psi_{problem}(\sigma,\delta) \equiv \sigma.robber.money = \delta(\sigma).robber.money \vee \sigma.vault.money = \delta(\sigma).vault.money$$

From the definition of the system, we see that this can be accomplished if either the vault is locked, the robber is drunk or the vault is empty. The maximal solution to $\Psi_{problem}$ is

$$\varphi_{max}(\sigma) \equiv drunk(\sigma.robber) \vee locked(\sigma.vault) \vee \sigma.vault.money = \emptyset$$

As a practical matter, a more restrictive solution will suffice

$$\varphi_{solve}(\sigma) \equiv locked(\sigma.vault)$$

that is, the problem is solved by constraining the initial states to those in which the vault is locked, without regard to whether the vault is empty and certainly without depending upon drunk bank robbers.

[Harrison, Ruzzo & Ullman 75] as well as this author (unpublished) have shown that even under very special circumstances the safety problem in protection matrix systems is undecidable. In our terminology, that means that there is no algorithm that can determine φ_{max}, the least restrictive solution, even for static problems.

Undecidability is not so negative a result as it might seem. φ_{max} is after all a maximal solution. In general, the user of a protection system is not seeking a maximal solution, but rather any reasonable solution that will solve the problem. In the bank robbery example, a reasonable solution meant locking the vault. In the Salary file example, an administrator might not really care about the obscure circumstances under which Cohen might be prevented from gaining write access to the Salary file. She is really only interested in showing that a particular solution will prevent Cohen's access.

The question, "What makes a solution reasonable?" will be discussed in more detail in section 7.3.

----- Section 4.4 --- A Methodology for Solving Problems

We may solve a behavioral problem by imposing an initial constraint Φ_{solve} and adding a mechanism M to a system. Instead of determining them together, we may find find the following approach more convenient: We first pick Φ_{solve}. Imposing Φ_{solve} will eliminate many of the unacceptable behaviors. Next determine M so that its addition eliminates the remaining unacceptable behaviors. Formally, define

>> Def 4-4] $\langle \Phi_{solve}, \Psi_{solve} \rangle$ enforces $\Psi_{problem}$ iff

$$\Phi_{solve}(\sigma) \supset (\forall H)(\Psi_{solve}(\sigma,H) \supset \Psi_{problem}(\sigma,H))$$

That is, Φ_{solve} only eliminates all unacceptable behaviors if the behaviors are already constrained by Ψ_{solve}. Once Ψ_{solve} is determined, a mechanism must be provided to enforce Ψ_{solve}. Our expectation is that enforcement of Ψ_{solve} is less complex (or can be accomplished more easily or more reasonably) than enforcement of $\Psi_{problem}$. Φ_{solve}, together with the mechanism that enforces Ψ_{solve} can then be shown to enforce $\Psi_{problem}$. Formally

Theorem 4-2]

If $\langle \Phi_{solve}, \Psi_{solve} \rangle$ enforces $\Psi_{problem}$
and $\langle \Phi_{solve}, M \rangle$ enforces Ψ_{solve}

then $\langle \Phi_{solve}, M \rangle$ enforces $\Psi_{problem}$

That is, after determining the remaining unacceptable behavior, find a mechanism M whose addition prevents those behaviors. That mechanism, along with Φ_{solve}, then solves the original problem.

For static problems we may similarly define

>> Def 4-5] $\langle \Phi_{solve}, \Psi_{solve} \rangle$ enforces $\Phi_{problem}$ iff

$$\Phi_{solve}(\sigma) \supset (\forall H)(\Psi_{solve}(\sigma,H) \supset \Phi_{problem}(H(\sigma)))$$

Theorem 4-3]

If $\langle \Phi_{solve}, \Psi_{solve} \rangle$ enforces $\Phi_{problem}$
and $\langle \Phi_{solve}, M \rangle$ enforces Ψ_{solve}
and M is homomorphic

then $\langle \Phi_{solve}, M \rangle$ enforces $\Phi_{problem}$

The technique discussed here is useful even when the mechanism is already specified, as long as only the mapping τ_M is specified, and not the initial constraint on the mechanism state Φ_M. After finding a $\langle \Phi_{solve}, \Psi_{solve} \rangle$ that enforces $\Phi_{problem}$ (or $\Psi_{problem}$), we may try to find an appropriate Φ_M so that the mechanism M, now fully specified, eliminates the remaining unacceptable behavior specified by Ψ_{solve}.

----- Section 4.5 --- Protection and Control

In this section, we will apply the methodology discussed in the previous section to solve a behavioral problem. We will consider a system that contains not one, but two mechanisms, a protection mechanism as well as a multiprogram control mechanism (section 3.5).

Consider a system with two kinds of objects, processes and files, and a single generic operation

copy(x,β,α): if process(x) ∧ file(β) ∧ file(α)
then β ← α

where Executor(copy(x,β,α)) = x

Let us suppose that a mechanism based on an access matrix is added to this system. The augmented system instead provides the generic operation

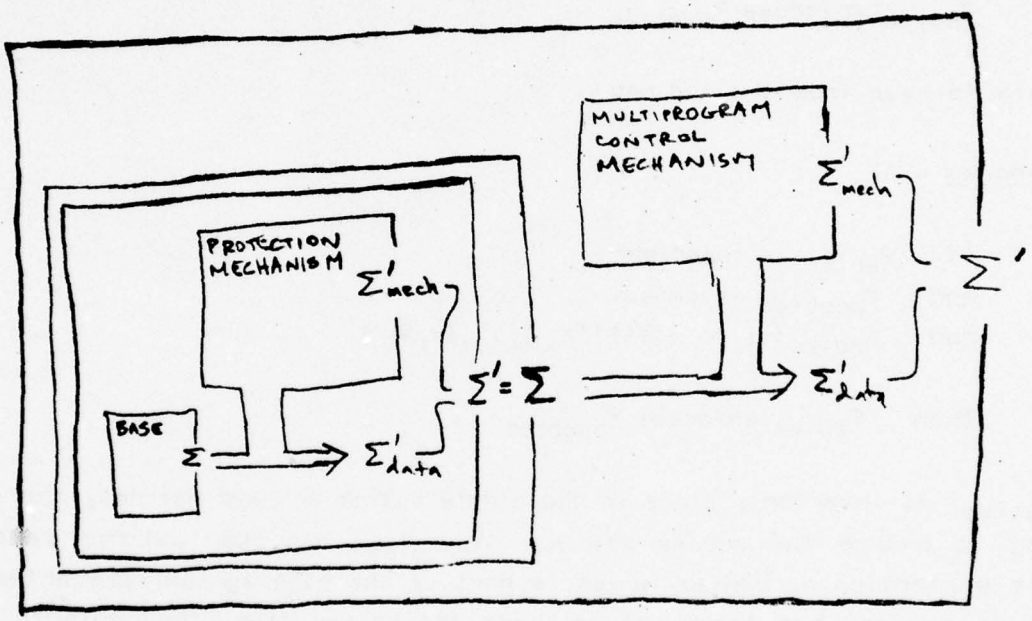
copy(x,β,α): if r ∈ <x,α> ∧ w ∈ <x,β>
∧ process(x) ∧ file(β) ∧ file(α)
then β ← α

That is, the mechanism checks to see whether x has the right to read α and write β before copying α to β.

We now treat this augmented system as the given base system and add a multiprogram control mechanism to it. The base system supplies the set of operations

```
copy(x,β,α):  if  r ∈ <x,α>  ∧  w ∈ <x,β>
                  ∧  process(x)  ∧  file(β)  ∧  file(α)
              then  β ← α
```

that is, what we called copy' above.



The multiprogram control mechanism M (as defined in section 3.5) prevents arbitrary execution of operations. The control mechanism state associates with each process a program counter (pc) and a code component containing the operations to be executed by the process. The mechanism guarantees that operations are executed in the base system only when pointed to by the pc. That is, if Executor(δ) is x, then x can execute δ in state σ' only if

```
σ'.x.code[σ'.x.pc]  =  "δ"
```

Now, suppose that we want to solve the following behavioral problem

$$(\text{markov})\ \Psi_{problem}(\sigma,\delta) \equiv \sigma.Salary = \delta(\sigma).Salary$$

that is, guarantee that the value of the Salary file remains unchanged over execution of any operation. The most direct way to solve this problem is to ignore the fact that there is a control mechanism augmenting the system, and simply guarantee that no process has the right to write the Salary file. That is, define

$$\Phi_{solve}(\sigma) \equiv (\forall x)(\ w \notin \langle x, Salary\rangle(\sigma)\ )$$

We can easily prove that

Φ_{solve} enforces $\Psi_{problem}$

This follows from the theorem

Theorem 4-4]

If Φ_{solve} is invariant
and $\Psi_{problem}$ is markov
and $\Phi_{solve}(\sigma) \supset (\forall\delta)(\ \Psi_{problem}(\sigma,\delta)\ )$

then Φ_{solve} enforces $\Psi_{problem}$

Φ_{solve} is invariant, since in the simple system we have defined, there is no way to change the access matrix. $\Psi_{problem}$ has been defined as markov. The protection mechanism (which is part of the base system) guarantees that if no process has the right to write the Salary file, the contents of the Salary file cannot be changed.

Suppose that for some reason, this solution is not allowed, and the best we can manage is the following

$$\Phi_{solve}(\sigma) \equiv (\forall x \neq Administrator)(\ w \notin \langle x, Salary\rangle(\sigma)\ )$$

That is, the system administrator may not be prevented from having permission to write the Salary file. The problem may be solved only if the administrator does not exercise this right - that is, does not write the Salary file. Formally this property can be stated as

$$(\text{markov})\ \Psi_{solve}(\sigma,\delta) \equiv (\forall a)(\ \delta \neq \text{copy}(\text{Administrator},\text{Salary},\alpha)\ )$$

Then, since copy is the only operation defined in this system, we can show that

$$\langle \Phi_{solve}, \Psi_{solve} \rangle \text{ enforces } \Psi_{problem}$$

Since the base system is augmented with a multiprogram control mechanism, Ψ_{solve} can be enforced by guaranteeing that the Administrator's code component does not contain an operation whose execution would alter file. If τ_M is the multiprogram control mechanism mapping defined in section 3.5 and Φ_M is defined as

$$\Phi_M(\sigma') \equiv (\forall \alpha, i)(\ \sigma'.\text{Administrator.code}[i] \neq \text{"copy(Administrator,Salary,}\alpha\text{)"}\ )$$

then M enforces Ψ_{solve}.

So, we first imposed an initial constraint Φ_{solve} that, by itself, did not enforce $\Psi_{problem}$. Rather, imposition of Φ_{solve} still permitted the occurrence of certain unacceptable behaviors (those not satisfying Ψ_{solve}). These remaining unacceptable behaviors could then be eliminated by specifying a constraint Φ_M on the mechanism state that guaranteed that M could enforce Ψ_{solve}.

In a sense this example is too simple. An application of the methodology that would be more clearly useful might use a Ψ_{solve} that was not markov. Such an example is left to future research.

----- Section 4.6 --- *** Constraining the Augmented System

In section 4.4, we showed that if the mapping τ_M of a mechanism is already provided, then a behavioral problem can be solved by imposing both an initial constraint Φ_{solve} on the base system, and an initial constraint Φ_M on the mechanism state. Since the user is presented with an augmented system, combining both the base system and the mechanism, it may be useful

to specify both constraints as the single constraint Φ'_{solve} which constrains the initial state of the augmented system.

In this section, we will show that any constraint Φ'_{solve} on the initial state of the augmented system can be decomposed into Φ_M and Φ_{solve}. The possibility of decomposition is not obvious for two reasons.

1. While Φ'_{solve} constrains the augmented system state, Φ_{solve} constrains the base system state. The mapping between them, τ_M, need not be trivial.

2. Φ_M must satisfy the property

$$\{ \tau_M(\sigma') \mid \Phi_M(\sigma') \} = \{ \tau_M(\sigma') \}$$

We shall now show that any Φ' can be decomposed into a Φ_M and a Φ such that $\Phi' = M:\Phi$.

Theorem 4-5)

Define Φ so that
$\Phi(\sigma) \equiv \sigma \in \{ \tau_M(\sigma') \mid \Phi(\sigma') \}$

Define Φ_M so that
$\Phi_M(\sigma') \equiv \Phi'(\sigma') \vee \neg\Phi(\tau_M(\sigma'))$

Then $\Phi' \equiv \Phi:M$
and $\{ \tau_M(\sigma') \mid \Phi_M(\sigma') \} = \{ \tau_M(\sigma') \}$

For example, the solution to the problem in the previous section can be written as

$$\Phi'_{solve}(\sigma') \equiv (\forall\alpha, i)(\sigma'.\text{Administrator.code}[i] \neq \text{"copy(Administrator,Salary,}\alpha\text{)"}) \wedge (\forall x \neq \text{Administrator})(w \notin \langle x, \text{Salary}\rangle(\sigma'))$$

which was decomposed as

$$\varphi_M(\sigma') \equiv (\forall \alpha, i)(\sigma'.\text{Administrator.code}[i] = \text{"copy(Administrator,Salary,}\alpha\text{)"})$$

$$\varphi_{solve}(\sigma') \equiv (\forall x \neq \text{Administrator})(u \notin \langle x, \text{Salary} \rangle(\sigma))$$

so that $\varphi'_{solve}(\sigma') = \varphi_M(\sigma') \wedge \varphi_{solve}(\tau_M(\sigma'))$

Chapter 5 - A Case Study

----- Section 5.1 --- Introduction

This chapter provides a case study of the solution to the following protection problem: Imagine that some system contains a set of sensitive objects. These objects are to be altered only by programs which have been verified to treat these objects properly.

We will solve this problem for the base system described in Appendix A. [Readers unfamiliar with capability-based protection systems are advised to turn to appendix A before proceeding] It is described as containing a capability-based protection mechanism. Each object in the system has two components, a C-list and a Value-part. The C-list of an object indicates which other objects may be accessed by (when it represents an executor) or through it. The Value-part of an object holds arbitrary data. In the case of objects which represent executors, the Value-part contains a representation of the program executed by that object. The system also models dynamic creation of objects, including (via the "call" operation) creation of new executors.

The remainder of this chapter is organized in the following way: We first specify the problem formally as a behavioral enforcement problem in notation independent of the mechanism provided. Next, using the definition of the mechanism, we specify a static problem that is easier than the given problem (any solution to the easier problem solves the given problem as well). We develop three increasingly general solutions to the problem. Finally we discuss the nature of solutions to protection problems and the impact of exercises such as this one on the design of protection mechanisms. In particular, we we may find that additional mechanisms are required or desirable in order to solve problems or to enhance reliability.

----- Section 5.2 --- The Problem

We may formally state the problem described above as follows:

(markov) $\Psi_{problem}(\sigma, \delta) \equiv$
$\neg Trusty(Executor(\delta)) \supset (\forall \beta)(Sensitive(\beta) \supset . \sigma.\beta = \delta(\sigma).\beta)$

[Instead of $\sigma.\beta = \delta(\sigma).\beta$, we might have written
$\neg Wacc(\beta, \sigma, \delta)$ as in section 4.2]

where

Trusty(x) means executor x is trusted to only execute programs verified to treat sensitive objects properly.

Sensitive(β) means β is a sensitive object. We shall assume that sensitive objects may not be executors.

The treatment of objects as Trusty or Sensitive is external to the system. No object itself contains any information indicating how we (as theorem provers) will treat it.

Treated as an enforcement problem, the formal specification above requires that no operation executed by a program that has not been verified be allowed to alter the contents of a sensitive object. It is important to note that this specification relies in no way upon the representation of the protection system.

We will actually solve an easier problem. But first we must note two facts about the system described in appendix A.

1. All operations are specified so that no executor x may alter another object β unless x has permission to write β - that is, $w \in \langle x, \beta \rangle$.

2. No object x may act as an executor unless $s \in \langle x, x \rangle$. No operation allows sharing of an "s"-right.

Therefore, we can solve the problem by guaranteeing that no non-trusty executor may ever have write access to a sensitive object. Actually we will solve an easier problem. We will show how to guarantee that no non-trusty executor has any access to sensitive objects. [We will discuss in section 5.8 how one might adapt the last solution given so that reads, but not writes, can be permitted.] This problem is, in effect, the Hidden Facilities problem (section 1.3). Sensitive objects represent the facilities that are to be hidden from certain users. Trusty executors represent those users who are trusted to use those facilities. Formally, we will be solving the problem

$$\text{(markov) } \Psi_{problem}(\sigma,\delta) \equiv \neg Trusty(Executor(\delta)) \supset (\forall\beta)(\, Sensitive(\beta) \supset \neg Acc(\beta,\sigma,\delta)\,)$$

where $Acc(\beta,\sigma,\delta)$ means that β is accessed as the result of executing δ in state σ.

We will convert $\Psi_{problem}$ to the static problem $\Phi_{problem}$, defined as

$$\Phi_{problem}(\sigma) \equiv (\forall x,\beta,q)\,(\, s \in \langle x,x\rangle(\sigma) \wedge q \in \langle x,\beta\rangle(\sigma)\,) \supset .\ Sensitive(\beta) \supset Trusty(x)\,)$$

Formally this substitution of problems can be justified by theorem 4-1 - that is

If Φ_{solve} enforces $\Phi_{problem}$
and $\Psi_{problem}$ is markov
and $\Phi_{problem}(\sigma) \supset (\forall\delta)(\, \Psi_{problem}(\sigma,\delta)\,)$

then Φ_{solve} enforces $\Psi_{problem}$

Below, we prove that $\Phi_{problem}(\sigma) \supset (\forall\delta)(\, \Psi_{problem}(\sigma,\delta)\,)$

By inspection of the operations defined in Appendix A, we find

$$x = Executor(\delta) \wedge Acc(\beta,\sigma,\delta) \supset .\ s \in \langle x,x\rangle(\sigma) \wedge (\, x = \beta \vee q \in \langle x,\beta\rangle(\sigma)\,)$$

That is, an executor can only access itself or some other object for which it has some access right. If we assume that $\varphi_{problem}(\sigma)$ holds, then we can show that

$$x = Executor(\delta) \wedge Acc(\beta, \sigma, \delta) \supset. \; Sensitive(\beta) \supset (x = \beta \vee Trusty(x))$$

Since executors cannot be sensitive

$$x = Executor(\delta) \wedge Acc(\beta, \sigma, \delta) \supset. \; Sensitive(\beta) \supset Trusty(x)$$

which is just a rearrangement of $\Psi_{problem}(\sigma, \delta)$

To find a φ_{solve} that enforces $\varphi_{problem}$, we might find a φ_{solve} that is invariant and stricter (though hopefully not much stricter) than $\varphi_{problem}$. Formally

Theorem 5-1]

If $\varphi_{solve} \subseteq \varphi_{problem}$
and φ_{solve} is invariant

then φ_{solve} enforces $\varphi_{problem}$

If such a solution could be found, then the protection mechanism would be shown to provide all the tools necessary to solve the problem. However, we will see below that additional constraints on the behavior of the system are required, which means that some additional mechanism must be added. That mechanism is assumed to be a control mechanism (as in section 3.5) which must be constrained so as to further specify the behavior of trusty executors. We must find a pair $\langle \varphi_{solve}, \Psi_{solve} \rangle$ such that Ψ_{solve} can be enforced by the control mechanism (section 4.4) and such that

$\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$

As above, we can pick φ_{solve} to be contained in $\varphi_{problem}$. If a behavioral constraint Ψ_{solve} is required, then, while we cannot e pect

φ_{solve} to be invariant (else Ψ_{solve} would not be needed at all), we can expect it to be Ψ_{solve}-invariant. That is, φ_{solve} remains satisfied so long as only histories satisfying Ψ_{solve} are executed. Formally,

>> Def 5-1] φ is Ψ-invariant iff

$$\Psi(\sigma, H) \supset. \ \varphi(\sigma) \supset \varphi(H(\sigma))$$

Theorem 5-2]

If $\varphi_{solve} \subseteq \varphi_{problem}$
and φ_{solve} is Ψ_{solve}-invariant

then $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$

To demonstrate Ψ_{solve}-invariance, we will find the following theorem useful:

Theorem 5-3]

If Ψ is markov
and $\Psi(\sigma, \delta) \supset. \ \varphi(\sigma) \supset \varphi(\delta(\sigma))$

then φ is Ψ-invariant

Throughout the remainder of this chapter, we will choose Ψ_{solve}'s that are markov and we will show that $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$ by showing that $\Psi_{solve}(\sigma, \delta) \supset. \ \varphi_{solve}(\sigma) \supset \varphi_{solve}(\delta(\sigma))$.

----- Section 5.3 --- Creation Rules

In the system described in appendix A, new objects may be created by the operations "call" and "create". Formally, it is best to treat new objects as if they were not actually created, but rather, were selected from a pool of existing objects that have not yet been used. Therefore, we assume that no object has access to any of these objects prior to the time of their selection. We express this formally by the following Creation Rule.

$R_{(new)}$: $(\forall q,x)(\ q \notin \langle x,N\rangle(\sigma)\ )$
where N made new in state σ

The description of the "create" operation indicates that the executor of the "create" will have access to the new object after execution of the operation. That is, if δ is [create(x)], and N is the object created in state σ, then

$\langle x,N\rangle(\delta(\sigma)) = \{r,w,c\}$

"Sensitive" is a predicate applied to object names, thus when a new object N is "created", we must decide whether it is to be treated as sensitive or not. It is clear that we must not treat it as sensitive if it is created by an untrusty executor x, for if N is the name of the object "created", then $w \in \langle x,N\rangle$ after creation. If N were treated as sensitive, but x were not trusty, then $\varphi_{problem}$ would immediately be violated. We define our treatment of created sensitive objects by the following Creation Rule.

$R_{(sensitive:\ create)}$: Sensitive(N) $\supset$ Verified(σ.x)
where N made new for
create(x) in state σ

We have noted that sensitive objects are not to be executors. Formally as part of each solution below, we will require that $\varphi_{sensitive}$ hold, where

$\varphi_{sensitive}(\sigma) \equiv$ Sensitive(β) $\supset$ s $\notin \langle\beta,\beta\rangle(\sigma)$

Since the new object created by the "call" operation is an executor, and sensitive objects may not be executors, there is no corresponding Creation Rule for "call". Initially, we will assume that there is a fixed number of trusty executors, so we need no creation rules for trusty objects at all.

----- Section 5.4 --- Verified Programs

A verified program is one that has been guaranteed to treat sensitive objects in some trusty way, that for the most part, need not concern us. We will only be concerned with those properties that verified programs need satisfy in order to solve the problem stated above - that is, guarantee that only verified programs can be used to alter sensitive objects.

While "Trusty" is a property of objects (their names, actually), "Verified" is a property of the contents of an object. We will write Verified(σ.x) if object x contains a verified program in state σ. We build a bridge between these two properties by guaranteeing that trusty executors contain verified programs.

$$\varphi_{verif}(\sigma) \equiv Trusty(x) \supset Verified(\sigma.x)$$

This requirement will be part of each of the solutions we will discuss below. Since we will be proving the invariance of those solutions, we note two properties of the "Verified" predicate which follow from the representation of "programs" in the system described in appendix A, which will be helpful in proving that invariance.

VRF1: $\sigma 1.\beta 1.Value\text{-}part = \sigma 2.\beta 2.Value\text{-}part$
$\supset.\quad Verified(\sigma 1.\beta 1) = Verified(\sigma 2.\beta 2)$

That is, the static representation of a program is contained wholly within the Value-part of an object, so if the Value-parts of two objects are the same, one contains a verified program only if the other one does.

We also note that execution of a verified program does not alter the property of its being verified (i.e. by changing the Value-part of the object containing the program). Formally

VRF2: $Trusty(x) \wedge x = Executor(\delta) \supset.$
$Verified(\sigma.x) \supset Verified(\delta(\sigma).x)$

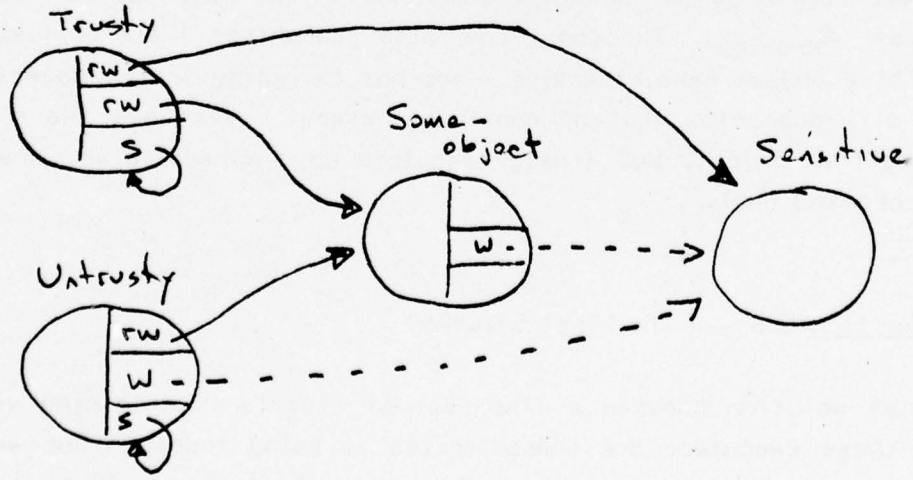
But what if x is not the executor of δ? Could execution of an operation by some executor alter the contents of some other object that is trusty? We

will arrange each of the solutions below so that such a situation is prevented.

That prevention is not difficult to accomplish. We noted above that an executor can only alter another object if it has permission to write it. Because of rule $R_{(new)}$ (and the semantics of "call"), when a *new executor* is created as the result of a call, no object, including the caller, is given permission to write the new executor. If we guarantee that no object initially has write access to trusty objects (as all solutions below will - via $\varphi_{trusty1}$ and $\varphi_{trusty2}$), then we can guarantee that if a trusty object initially holds a verified program, then it will continue to hold a verified program after execution of any operation, for its Value-part cannot be altered.

Next we will examimne whether there are any actions a trusty executor might perform that could violate $\varphi_{problem}$. If there are any, then we must establish additional conditions that verified programs must satisfy.

A trusty executor might grant access for a sensitive object to some object also accessable to an untrusty executor. If the untrusty executor subsequently takes that access, then $\varphi_{problem}$ would be violated, for an untrusty executor could gain access to a sensitive object. This is depicted in the diagram below.



$grant_q$(Trusty,Some-object,Sensitive) (q ∈ {r,w,c } must be prevented. A similar violation may occur if a trusty executor executes a "call"

operation, passing as an argument, access to a sensitive object (e.g. call(Trusty,Some-object,Sensitive)).

We must now imagine that some control mechanism augments the protection system provided, for these violations may not be prevented by constraining the initial state of the access matrix alone. It is necessary to guarantee that trusty executors do not execute the operations described above. That guarantee can be formalized as constraint on the behavior of verified programs.

We assume that the control mechanism state can be so constrained (remembering from section 3.4 that a constraint on the control state models specifications for the programs executed) as to prevent the occurrence of the operations noted above when executed by objects containing verified program. Formally, we expect that such a constraint can enforce the following behavioral constraint in the base system (which includes the protection mechanism):

$$\begin{aligned}&\text{(markov) } \Psi_{solve}(\sigma,\delta) \equiv \\ &\quad \text{Verified}(\sigma.x) \wedge x = \text{Executor}(\delta) \wedge \text{Sensitive}(\beta) \\ &\quad \supset. \quad \delta \neq \text{grant}_q(x,\alpha,\beta) \wedge \delta \neq call(x,\alpha,\beta) \\ &\qquad\qquad \wedge \delta \neq \text{call}(x,\beta,\alpha)\end{aligned}$$

That is, verified programs do not execute operations which could lead to a violation of $\Phi_{problem}$. The constraint also guarantees ($\delta \neq \text{call}(x,\beta,\alpha)$) that sensitive object remain passive - are not called by trusty objects (our solutions will guarantee that no non-trusty executor ever has the right to call a sensitive object, but trusty executors do, in particular, when they have just created one).

----- Section 5.5 --- The First Solution

The first solution assumes a fixed set of executors containing verified programs; these executors are characterized as being trusty. Our solution first requires that $\Phi_{sensitive}$ and Φ_{verif} be satisfied - that is, there can be no sensitive executors, and each trusty object must contain a verified program. We noted that Φ_{verif} could remain invariant only if initially, a trusty executor could be accessed by no other object. Formally

$$\varphi_{trusty1}(\sigma) \equiv Trusty(x) \supset (\forall q \neq s, \alpha)(\, q \notin \langle \alpha, x \rangle(\sigma)\,)$$

Now we must only guarantee that only trusty objects may directly access sensitive objects.

$$\varphi_{direct}(\sigma) \equiv (\forall q)(\, q \in \langle x, \beta \rangle(\sigma) \supset .\ Sensitive(\beta) \supset Trusty(x)\,)$$

The solution to $\varphi_{problem}$ is the conjunction of these predicates.

$$\varphi_{solve} \equiv \varphi_{sensitive} \wedge \varphi_{verif} \wedge \varphi_{trusty1} \wedge \varphi_{direct}$$

$\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$ since

$$\varphi_{solve} \subseteq \varphi_{direct} \subseteq \varphi_{problem}$$

and φ_{solve} is Ψ_{solve}-invariant which we prove by showing that $\Psi_{solve}(\sigma, \delta)$ and $\varphi_{solve}(\sigma)$ guarantee $\varphi_{solve}(\delta(\sigma))$ (Theorems 5-2 and 5-3).

1. $\varphi_{sensitive}(\delta(\sigma))$. There is simply no way to create new sensitive executors.

2. $\varphi_{verif}(\delta(\sigma))$. If Trusty(x), then by $\varphi_{trusty1}$, x cannot be written into by another executor of δ through execution of δ. φ_{verif} and VRF2 guarantee that x cannot alter verified-ness by its own action. Finally, no new trusty objects can be created.

3. $\varphi_{trusty1}(\delta(\sigma))$. If Trusty(x), then by $\varphi_{trusty1}$, no other object has access to x. The operations of Appendix A permit access to be granted or taken, but access to x cannot be gained if no object initially has access to x. Finally, no new trusty objects can be created.

4. $\varphi_{direct}(\delta(\sigma))$. Suppose β is sensitive and after execution of δ, a non-trusty object x gained access to β. This could not happen. Consider the following possibilities:

--- a) A trusty executor granted x access to β. By φ_{verif}, the executor contains a verified program. Prevented by Ψ_{solve}.

--- b) x took access to β from some object which must be trusty due to φ_{direct}. Prevented by $\varphi_{trusty1}$, which guarantees that no object can have read access to a trusty object, required by the semantics of "take".

--- c) x created a sensitive object. Prevented by $R_{(sensitive:\ create)}$, since x is not trusty.

--- d) β was called, resulting in a new trusty executor x with read access to β. By φ_{direct}, the caller must be trusty. Prevented by (φ_{verif} and) Ψ_{solve}.

--- e) β was passed as an argument when x was created as the result of a call. By φ_{direct}, the caller must be trusty. Prevented by (φ_{verif} and) Ψ_{solve}.

Q.E.D.

----- Section 5.6 --- The Second Solution

For our second solution, we will not presume that trusty objects are executors only. They may be non-executors that can be "called", resulting in the creation of new trusty executors. This implies that we must add a new Creation Rule.

$R_{(trusty:\ call)}$: Trusty(N) iff Trusty(β)
where N made new for
call(x,β,α) in state σ

Even when an object is not an executor, its Value-part may hold the representation of a program. In fact, according to appendix A, when a "call" is made, the program to be executed by the newly created executor is copied from the Value-part of the object called. So, as in the first solution, all trusty objects (whether or not they are executors) must

contain a representation of a verified program. That is, φ_{verif} must be satisfied.

We will continue to require that only trusty objects may access sensitive objects, so φ_{direct} must be satisfied as well.

Finally, though we permit trusty objects to be called, they may not be read from (lest some other object take access to a sensitive object) nor be written into (lest some untrusty executor alter the representation of the verified program in the Value-part).

$$\varphi_{trusty2}(\sigma) \equiv Trusty(\beta) \supset. \quad r \notin \langle\alpha,\beta\rangle(\sigma) \wedge w \notin \langle\alpha,\beta\rangle(\sigma)$$

Our solution is

$$\varphi_{solve} \equiv \varphi_{sensitive} \wedge \varphi_{verif} \wedge \varphi_{trusty2} \wedge \varphi_{direct}$$

As in the first solution, $\varphi_{solve} \subseteq \varphi_{direct} \subseteq \varphi_{problem}$. Below we prove that $\Psi_{solve}(\sigma,\delta)$ and $\varphi_{solve}(\sigma)$ guarantee $\varphi_{solve}(\delta(\sigma))$.

1. $\varphi_{sensitive}(\delta(\sigma))$. Same as for solution 1.

2. $\varphi_{verif}(\delta(\sigma))$. If trusty(x) and δ is not a call operation which creates x, then the proof is similar to that for solution 1. If δ is a call which creates x, then by $R_{(trusty: call)}$, the object called was trusty. By φ_{verif}, that object contained a a verified program. Since call copies the Value-part of that object to x, by VRF1, x contains a verified program as well. Finally, trusty objects cannot be created except by a call.

3. $\varphi_{trusty2}(\delta(\sigma))$. If trusty($\beta$) and δ is not a call operation which creates β, the proof is similar to that of trusty1 in solution 1. If δ is a call which creates β, then by $R_{(new)}$ and the semantics of call, no object gains read or write access to β. Finally, trusty objects cannot be created except by call.

4. $\varphi_{direct}(\delta(\sigma))$. Proof same as for solution 1.

Q.E.D.

----- Section 5.7 --- The Third Solution

In the previous two solutions, we have required that only trusty objects have direct access to sensitive objects. In our third solution, we will permit that access to be indirect. We permit capabilities for sensitive objects to be held in other objects. These other objects must be sensitive as well, and must not be executors (or callable), since the programs they might execute are not verified, and therefore, they might execute one of the operations prevented by Ψ_{solve}. Instead of requiring that Φ_{direct} hold, we require

$$\Phi_{indirect}(\sigma) \equiv q \in \langle\alpha,\beta\rangle(\sigma) \supset (\text{Sensitive}(\beta) \supset. \text{Trusty}(\alpha) \vee \text{Sensitive}(\alpha))$$

Our third solution, then is

$$\Phi_{solve} \equiv \Phi_{sensitive} \wedge \Phi_{verif} \wedge \Phi_{trusty2} \wedge \Phi_{indirect}$$

Before proving invariance, we note that since trusty objects may be called, and since non-trusty (though sensitive) objects can access sensitive objects, we may weaken the requirements for operations that may not be executed by trusty programs. We change Ψ_{solve} to require that

$$\begin{aligned}(\text{markov})\ \Psi_{solve}(\sigma,\delta) \equiv\ & \text{Verified}(\sigma,x) \wedge x = \text{Executor}(\delta) \wedge \text{Sensitive}(\beta) \\ \supset.\ & (\delta = \text{grant}_q(x,\alpha,\beta) \supset. \text{Trusty}(\alpha) \vee \text{Sensitive}(\alpha)) \\ \wedge\ & (\delta = \text{call}(x,\alpha,\beta) \supset \text{Trusty}(\alpha)) \\ \wedge\ & (\delta \neq \text{call}(x,\beta,\alpha))\end{aligned}$$

Now, we note that

$$\Phi_{solve} \subseteq (\Phi_{sensitive} \wedge \Phi_{indirect}) \subseteq \Phi_{problem}$$

So, again we need prove only that $\Psi_{solve}(\sigma,\delta)$ and $\Phi_{solve}(\sigma)$ together guarantee $\Phi_{solve}(\delta(\sigma))$.

1. $\varphi_{sensitive}(\delta(\sigma))$. Same as for solutions 1 and 2.

2. $\varphi_{verif}(\delta(\sigma))$. Same as for solution 2.

3. $\varphi_{trusty2}(\delta(\sigma))$. Same as for solution 2.

4. $\varphi_{indirect}(\delta(\sigma))$. Suppose that β is sensitive and the object α which is neither trusty nor sensitive has access to β aftr execution of δ. This cannot happen. The possibilities are:

--- a) A trusty executor granted α access to β. Prevented by (φ_{verif} and) Ψ_{solve}.

--- b) α took access to β from some object, which by $\varphi_{indirect}$ must be trusty or sensitive. Prevented by $\varphi_{trusty2}$ if that object is trusty, since "take"-ing requires read access. Prevented by $\varphi_{indirect}$ if that object is sensitive.

--- c) α created a sensitive object. Prevented by $R_{(sensitive:\ create)}$ since we assumed that α is not trusty.

--- d) $\beta$was called, resulting in a new executor α which has read access to β. By $\varphi_{indirect}$, the caller must either be sensitive (prevented by $\varphi_{sensitive}$) or trusty (prevented by φ_{verif} and Ψ_{solve}).

--- e) β was passed as an argument when α was created as the result of a call. By $\varphi_{indirect}$, the caller must be sensitive (prevented by $\varphi_{sensitive}$) or trusty, which by (φ_{verif} and) Ψ_{solve} would require that the object called be trusty as well. By $R_{(trusty:\ call)}$, α would then also be trusty which contradicts the assumption that α is neither trusty nor sensitive.

Q.E.D.

----- Section 5.8 --- Conclusion

In this chapter, we explored solutions to a protection problem that required both initial constraints on the base system (including an access matrix mechanism) as well as behavioral constraints enforced by constraining the programs executed by the control mechanism.

In our final solution, we constrained the protection state by requiring that sensitive objects could not be executors and could only be accessed by trusty objects and other sensitive objects. We further guaranteed that, while trusty objects might be called, no object might have have read or write access to one.

[New trusty objects can only be created through "call", and not by a "create". We might extend the solution to permit such creations, though only by trusty executors, who would then be permitted to have read and write access to those objects, presumably for the purpose of making new trusty programs. The reader might note that access rights for trusty objects would further complicate Ψ_{solve}, for we would have to guarantee that trusty executors would not grant these accesses to non-trusty objects (perhaps not grant them at all), and we would have to guarantee that in filling in the Value-part of a new trusty program, the executor would insure that it was verified]

[The reader might see how a solution could be found to the original problem, which permitted non-trusty executors to read, though not write, sensitive objects. We might designate some of the sensitive objects as "cautious" and permit non-trusty executors to have read, but no other access to these objects. Cautious objects might even have access rights for other sensitive objects, although only those which are also cautious, and only r-rights]

In each of our solutions, we found we had to constrain the behavior permitted by the control mechanism, by specifying a property that must be satisfied by verified programs. In solving a problem such as the one studied here, it is often desirable for the solution to constrain the protection state only. Proving that programs meet specifications is a task

one might in general like to avoid. Furthermore, unreliable hardware is more likely to violate security by altering the program than by altering the protection state, especially if the protection state is coded to permit error detection and correction.

If the problem studied here were important enough, one might try to solve it by adding an additional mechanism to the system so that the constraint on the control state might be eliminated.

In this case, we might consider marking each object, indicating when an object is trusty or sensitive. Whenever an object is created by a trusty executor, it is marked as sensitive. Whenever a trusty object is called, the new executor created is marked as trusty as well. The mechanism might then directly guarantee that only trusty executors may write into sensitive objects. Additional reliability may be obtained if this mechanism also prevents those grants and calls specified by Ψ_{solve}.

The problem we have been studying can be thought of as a simplification of a system containing multiple types of sensitive objects, each of which is to be written only by trusty executors of the same type. These executors may be thought of as the protected subsystem for that type. The mechanism suggested above is then seen to be analogous to one which stamps types on each object and permits an object to be written by an executor only if the type stamped on the executor matches the type stamped on the object. If this mechanism also prohibited those grants and calls specified by Ψ_{solve} as it would be altered to include the case of multiple types, then the reliability of the system might be further enhanced.

The major point to be gleaned from the discussion above is this: When a new system is designed, it is especially important that we undertake the solution of problems such as the one posed in this chapter. By so doing, we determine how suitable are the mechanisms provided by the system, and what additional mechanisms might be added to solve problems or to enhance reliability.

Chapter 6 - Productive Problems

----- Section 6.1 --- Enforcement Problems and Productive Problems

The behavioral problems described in the previous sections have all been enforcement problems. They have represented guarantees about the continuing behavior of the system. For example, "No untrusty executor must ever be able to write a sensitive object" and "The robber must never be able to get any money by robbing the bank". That is, all behaviors permitted in the system were required to be acceptable. For a productive problem, we must only guarantee that from any appropriate initial state, some acceptable behavior is possible. When a productive problem is represented statically - as a property of the state, $\varphi_{problem}$ - it is solved by guaranteeing that some behavior results in a state satisfying $\varphi_{problem}$. In addition to formally defining productive problems, we will show how the methodology of section 4.4 can be extended to them.

Protection problems may often be thought of as having two parts: How can some behavior be enforced (the enforcement problem), and under what conditions may the solution to that problem be imposed on the system (the productive problem). We will explore this situation in a formal setting in this chapter.

Finally, in a protection system, we may expect that some executor (or set of executors) is responsible for bringing about the conditions that guarantee the solution of a problem. Other executors may wish to thwart these responsible executors. We might like to guarantee that a protection mechanism permits the responsible executors to produce a solution even when other executors arbitrarily interfere. In this chapter, we will formally characterize such guarantees, as well as formally specifying solutions to productive problems.

----- Section 6.2 --- The Souffle Example

Consider the making of a spinach souffle. Our domain Σ is what might be called the kitchen state space. Each element $\sigma \in \Sigma$ represents some state of the kitchen - what ingredients and utensils are in the kitchen, what's being used, what is mixed, what is cooking, etc. The operations $\delta \in \Delta$ are state transitions for this state space. For example

$\delta 1(x,y,z)$: add x tsp y to bowl z
$\delta 2(x,y,z)$: mix x at speed y for time z
$\delta 3(x,y,z)$: bake x at temperature y for time z

The problem is: Make a spinach souffle. Unlike static problems, we don't want to guarantee that the system invariantly contains a spinach souffle (in fact, we rather expect that if the souffle is good it won't stay around long at all). Instead, starting in a state containing the appropriate ingredients, we want to transform the state and produce one containing a spinach souffle. This transformation is generally accomplished by following (executing) a recipe.

There may be more than one acceptable spinach souffle. We will write $\varphi_{problem}(\sigma)$ iff σ is a state containing an acceptable spinach souffle. We call $\varphi_{problem}$ a static productive problem.

A recipe is simply some sequence of cooking operations, that is a history, H_{recipe}.

The recipe requires that we start out with certain ingredients and utensils. This is a constraint on the initial state. There may be more than one acceptable initial state. For example, it may not matter much if the initial state has oleomargarine instead of butter or a 9-inch baking dish instead of a 10-inch baking dish. We write $\varphi_{solve}(\sigma)$ iff σ contains the necessary initial utensils and ingredients for the given recipe.

Given an appropriate initial state, the recipe presumably will produce an acceptable spinach souffle. That is

$$\varphi_{solve}(\sigma) \supset \varphi_{problem}(H_{recipe}(\sigma))$$

Of course, there may be more than one recipe that produces a spinach souffle, although some recipes will work only with certain initial ingedients and utensils and not with others. So we will write $\Psi_{solve}(\sigma, H_{recipe})$ iff H_{recipe} is a recipe that produces an acceptable spinach souffle when used with the ingedients and utensils supplied by state σ. Formally

$$\Phi_{solve}(\sigma) \supset (\forall H)(\Psi_{solve}(\sigma, H) \supset \Phi_{problem}(H(\sigma)))$$

The formula above is exactly the same formula as that for the solution to a static enforcement problem, $\Phi_{problem}$. That is,

$$<\Phi_{solve}, \Psi_{solve}> \text{ enforces } \Phi_{problem}$$

In the next section we will explore this similarity.

----- Section 6.3 --- Producing Solutions

In chapters 4 and 5, Ψ_{solve} was specified in such a way that (assuming that the system was initially constrained by Φ_{solve}) execution of any behavior satisfying Ψ_{solve} results in a state satisfying $\Phi_{problem}$. In the previous section Ψ_{solve} is specified in exactly the same way. However, in solving a behavioral problem we expect that any behaviors not satisfying Ψ_{solve} will be prevented by the mechanism augmenting the system; in solving a productive problem, we expect that the mechanism guarantees that some behavior satisfying Ψ_{solve} will be produced. Formally we define (compare with definition 2-19)

>> Def 6-1] $<\Phi, M>$ produces Ψ iff

$$(M:\Phi)(\sigma') \supset (\exists H')(\Psi(\tau_M<\sigma', H'>))$$

that is, if $<\Phi_{solve}, M>$ produces Ψ_{solve}, then for any initial state satisfying Φ_{solve}, the mechanism permits the production of at least one behavior that satisfies Ψ_{solve}. If M is a control mechanism (section 3.4), then Φ_M can be used to specify the program (recipe) that is to be executed.

[We previously showed (theorem 3-4) that all behavioral

constraints enforced by a decision mechanism are monotonic. This is not true of constraints produced by a decision mechanism. Ψ_{solve} may satisfy $\langle\sigma, \delta1\delta2\delta3\rangle$ in some system, but not $\langle\sigma, \delta1\delta2\rangle$. Certainly this makes sense for the spinach souffle example. While $(\delta1\delta2\delta3)(\sigma)$ might contain an edible souffle, $(\delta1\delta2)(\sigma)$ may not, especially if $\delta3$ is the operation "bake the souffle".]

We further can define $\langle\Phi_{solve}, M\rangle$ produces $\Phi_{problem}$ to mean that, from any initial state satisfying $\Phi_{problem}$, the mechanism can be used to produce a behavior whose result state satisfies $\Phi_{problem}$. If M is a control mechanism, Φ_M may specify a program whose execution will result in a state satisfying $\Phi_{problem}$ when executed in a state satisfying Φ_{solve}.

>> Def 6-2] $\langle\Phi_{solve}, M\rangle$ produces $\Phi_{problem}$ iff

$$(M{:}\Phi_{solve})(\sigma') \supset (\exists H')(\Phi_{problem}(\tau_M(H'(\sigma'))))$$

Thus the productive problem $\Phi_{problem}$ can be distinguished from the *enforcement problem* $\Phi_{problem}$ which is solved when

$$(M{:}\Phi_{solve})(\sigma') \supset (\forall H')(\Phi_{problem}(\tau_M(H'(\sigma'))))$$

At the end of the previous section, we noted that productive problems could be solved by a pair $\langle\Phi_{solve}, \Psi_{solve}\rangle$ that enforces $\Phi_{problem}$. This can be explained by the following theorem (compare with theorem 4-3)

Theorem 6-1]

If $\langle\Phi_{solve}, \Psi_{solve}\rangle$ enforces $\Phi_{problem}$
and $\langle\Phi_{solve}, M\rangle$ produces Ψ_{solve}
and M is homomorphic

then $\langle\Phi_{solve}, M\rangle$ produces $\Phi_{problem}$

Like a static enforcement problem, a static productive problem $\Phi_{problem}$ is shorthand for the behavioral problem $\Psi_{problem}$ defined so that

$$\Psi_{problem}(\sigma, H) \equiv \Phi_{problem}(H(\sigma))$$

that is, a behavior is acceptable (satisfying $\Psi_{problem}$) if the state resulting from its execution satisfies $\Phi_{problem}$.

Theorem 6-2]

If $\langle \Phi_{solve}, \Psi_{solve} \rangle$ enforces $\Psi_{problem}$
and $\langle \Phi_{solve}, M \rangle$ produces Ψ_{solve}

then $\langle \Phi_{solve}, M \rangle$ produces $\Psi_{problem}$

The two theorems above are analogous to those of section 4.4 and suggest the following methodology for solving productive problems:

1. Find an initial constraint Φ_{solve} which eliminates all states in which no acceptable behavior can be produced.

2. Characterize (Ψ_{solve}) a set of acceptable behaviors. Each state satisfying Φ_{solve} must be the initial state of at least one behavior in the set.

3. Determine a mechanism M that can produce at least one behavior characterized by Ψ_{solve} for each initial state satisfying Φ_{solve}.

----- Section 6.4 --- *Producing Solutions to Protection Problems*

What we tend to informally call a protection problem often turns out to be two problems, one of them productive. Suppose that we are interested in solving some enforcement problem $\Psi_{problem}$. We may find some solution Φ_{solve} that solves the problem. It is useful to know how Φ_{solve} may itself be produced.

For example, in section 4.3 we discussed the bank robbery problem.

$$(markov)\ \Psi_{problem}(\sigma, \delta) \equiv \sigma.robber.money = \delta(\sigma).robber.money$$
$$\sigma.vault.money = \delta(\sigma).vault.money$$

in a system in which

δ1: if ¬drunk(robber) ∧ ¬locked(vault)
(robber.money ← robber.money + vault.money;
vault.money ← 0)

We showed that $\Psi_{problem}$ could be enforced by a solution that required that either the vault be locked or empty, or that the robber be drunk. Further, we showed that a requirement that the vault simply be locked was an acceptable (though not minimal) solution.

$$\Phi_{solve}(\sigma) \equiv locked(\sigma.vault)$$

Another solution would simply require only that the robber be drunk. Intuitively, that solution is <u>not</u> acceptable. The reason is that such a solution may (presumably) not be produced.

We made the implicit assumption that someone (e.g. the bank manager) wanted to solve the problem and would take some action to bring about the solution. We imagine that locking the vault is within the control of the bank manager; guaranteeing drunk bank robbers (presumably) is not.

Formally we treat Φ_{solve} as a <u>productive problem</u> $\Phi^{*}_{problem}$.

$$\Phi^{*}_{problem} \equiv_{def} \Phi_{solve}$$

We expect that in the case that $\Phi^{*}_{problem}$ characterizes the locked vault, some other operation, for example

δ2: vault.lock ← tt

(note we can now define the predicate locked as
$locked(x) \equiv_{def} x.lock$)

can be executed to satisfy $\Phi^{*}_{problem}$. That is, we find that $\langle \Phi^{*}_{solve}, \Psi^{*}_{solve} \rangle$ enforces $\Phi^{*}_{problem}$ where

$$\Phi^{*}_{solve}(\sigma) \equiv tt$$
$$\Psi^{*}_{solve}(\sigma, H) \equiv H = \delta 2$$

In any state, execution of $\delta 2$ (locking of the vault, presumably by the bank manager, Executor($\delta 2$) = manager) will produce $\Phi^{\star}_{problem}$.

In the example above we could always solve $\Phi^{\star}_{problem}$. This may not always be the case. Possibly the vault could be locked only by someone with a key. Instead of a single command $\delta 2$, we might have

$\delta 2(x)$: if key $\in$ <x,vault> then vault.lock $\leftarrow$ tt
where Executor($\delta 2(x)$) = x

This introduces the possibility that no one may have the key. $\Phi^{\star}_{problem}$ can only be solved if someone has the key. For example, suppose that x has the key. $\Phi^{\star}_{problem}$ can then be solved by $\langle \Phi^{\star}_{solve}, \Psi^{\star}_{solve} \rangle$ where

$\Phi^{\star}_{solve}(\sigma) \equiv$ key $\in$ <x,vault>(σ)
$\Psi^{\star}_{solve}(\sigma) \equiv H = \delta 2(x)$

Now if

Φ_{solve} enforces $\Psi_{problem}$
$\Phi^{\star}_{problem} \equiv_{def} \Phi_{solve}$
$\langle \Phi^{\star}_{solve}, \Psi^{\star}_{solve} \rangle$ enforces $\Phi^{\star}_{problem}$

then we say that $\Phi^{\star}_{solve}$ is a context for the solution of $\Psi_{problem}$. In this example, it is possible to produce a state in which a bank robbery yields the robber no money in any context in which x has a key to the vault.

----- Section 6.5 --- Producing Mechanisms

In section 6.3 we showed that a solution $\langle \Phi^{\star}_{solve}, M^{\star} \rangle$ could produce $\Phi^{\star}_{problem}$ by guaranteeing that for any state satisfying $\Phi^{\star}_{solve}$, the mechanism $M^{\star}$ permits execution of some history resulting in a state satisfying $\Phi^{\star}_{problem}$. In section 6.4, we showed how $\Phi^{\star}_{problem}$ might itself be a solution to some enforcement problem $\Psi_{problem}$. That is

Φ_{solve} enforces $\Psi_{problem}$

where $\Phi_{solve} \equiv \Phi^{\star}_{problem}$

But, we have noted that an enforcement problem may be solved by adding a mechanism M as well. That is

$$\langle \Phi_{solve}, M \rangle \text{ enforces } \Psi_{problem}$$

The mechanisms M and M☆ may be almost, but not exactly, the same. τ_M and $\tau_{M☆}$ are the same. However, $\Phi_{M☆}$ represents the initial constraint on the mechanism state. After execution of some history (producing Φ_{solve}), $\Phi_{M☆}$ may no longer hold; instead Φ_M is expected to hold. Formally we define (compare with definition 6-2)

>> Def 6-3] $\langle \Phi☆, M☆ \rangle$ produces $\langle \Phi, M \rangle$ iff

$$\tau_{M☆} = \tau_M \wedge$$
$$(M☆ : \Phi☆)(\sigma') \supset (\exists H')((M:\Phi)(H'(\sigma')))$$

Thus if we want to guarantee not only that some enforcement problem $\Psi_{problem}$ can be solved, but that the solution can be produced by $\langle \Phi☆, M☆ \rangle$, then we must find a $\langle \Phi, M \rangle$ such that

$\langle \Phi, M \rangle$ enforces $\Psi_{problem}$ and
$\langle \Phi☆, M☆ \rangle$ produces $\langle \Phi, M \rangle$

----- Section 6.6 --- Local Solutions

Not
everyone in a given system (e.g. the bank robber) necessarily wants a problem to be solved. We say that a problem can be locally produced by some set of users if an acceptable behavior can be produced in the face of active interference by other users. We will motivate the formalization of this relation by continuing the discussion of the bank robbery example (section 6.4)

Suppose that x has a key, but x is not the manager (perhaps x is the manager's goat-like dog who ate the key). Theoretically $\Phi☆_{problem}$ (which requires that the bank vault be locked) can be solved, but possibly not by the executor (in this case the bank manager) that wants it solved. We really are only interested in the solution

$$\Phi^{*}_{solve}(\sigma) \equiv key \in \langle manager, vault\rangle(\sigma)$$
$$\Psi^{*}_{solve}(\sigma, H) \equiv H = \delta 2(manager)$$

which guarantees that the manager is the executor that can produce the desired solution.

First let us imagine that the system contains yet one more command.

$\delta 3(x,y)$: if evil(x) $\wedge$ forgetful(y)
then $\langle y, vault\rangle \leftarrow \langle y, vault\rangle - key$

where Executor($\delta 3(x,y)$) = x

That is, if y is forgetful, y may forget her key to the vault and the key may be thrown away by the evil executor x (remember that the key does not unlock, but only locks, the vault. Perhaps a different key may be required to open it). If the manager forgets her keys and the evil Demono throws them away (executes $\delta 3$(Demono,manager)), the vault cannot be locked.

Now formally, the solution given above can still produce a state satisfying $\Phi^{*}_{problem}$, however, Ψ^{*}_{solve} does not take into account any operations other than $\delta 2$(manager). We should like to find a solution that guarantees that even if other executors interfere, $\Phi^{*}_{problem}$ can still be satisfied. For example, we might strengthen Φ^{*}_{solve}.

$$\Phi^{*}_{solve}(\sigma) \equiv key \in \langle manager, vault\rangle(\sigma) \wedge \neg forgetful(\sigma, manager)$$

That is, as long as the manager is not forgetful, it does not matter what other executors do. Once $\delta 2$(manager) executes, the problem is solved.

In general, we might imagine that if some consortium of executors X were trying to bring about a state satisfying $\Phi^{*}_{problem}$, they might be thwarted by other executors. This might be prevented in two ways. The approach taken above strengthens the initial constraint Φ^{*}_{solve} so that actions taken by other executors can have no ill effects. Alternately, the mechanism M^{*} producing Ψ^{*}_{solve} might be so constrained (or chosen) to prevent operations of other executors that would thwart satisfaction of $\Phi^{*}_{problem}$.

In either case, we want to guarantee that if $\langle \Phi\star_{solve}, M\star \rangle$ permits execution of a history that produces a state satisfying $\Phi\star_{problem}$, then adding or removing operations from that history not executed by executors in X should have no effect on the satisfaction of $\Phi\star_{problem}$.

Below, we will formally define H/X to mean H with all operations removed except those executed by X. As a result, H1/X = H2/X if the sequence of operations performed by executors in X are the same for histories H1 and H2. So, we can define

>> Def 6-4] $\langle \Phi\star, M\star \rangle$ locally produces $\langle \Phi, M \rangle$ for X iff

$$\tau_{M\star} = \tau_M \quad \wedge$$
$$(M\star : \Phi\star)(\sigma') \supset (\exists H')(\forall [\, H\star/X = H'/X \,])(\, (M:\Phi)(H\star(\sigma')) \,)$$

Below we define H reduced by X. The similarity in notation to that of definition 3-10 is justified by a similarity in intent ("reduction"), as well as by a similarity in the form of the definitions.

>> Def 6-5] H/X (recursively defined)

$$\lambda/X \iff \lambda$$

$$(H\delta)/X \iff$$
Let R be H/X in
if Executor(δ) $\in$ X then Rδ else R

We close this section by pointing out the likely meaning of $\Phi_{M\star}$ when $\langle \Phi\star, M\star \rangle$ locally produces $\langle \Phi, M \rangle$ for X in the case that M\star is a multiprogram control mechanism (section 3.5). $\Phi_{M\star}$ presumably specifies the program components of X whose execution (when the base state satisfies $\Phi\star$) results in a state satisfying M:Φ.

----- Section 6.7 --- Examples of Productive Problems

In previous chapters we have shown how enforcement problems correspond to many protection and synchronization problems. In section 6.4 we noted that productive problems may arise by considering how solutions to those enforcement problems may be brought about. In this section, we will provide examples if productive problems important in and of themselves.

First we consider the static problem

$$\Phi_{problem}(\sigma) \equiv w \notin \langle Cohen, Salary \rangle(\sigma)$$

In section 4.2 we treated this problem as an enforcement problem - as a property that had to be enforced. As an enforcement problem, $\Phi_{problem}$ corresponds to the Access problem - guarantee that Cohen can never gain access to the Salary file.

When we treat $\Phi_{problem}$ as a productive problem - as a property to be produced, it corresponds to an entirely different problem, a Revocation problem [Redell 74]. $\Phi_{problem}$ is solved (as a productive problem) by determining how the system may be brought to a state in which Cohen does not have write access to the Salary file. If $\langle \Phi_{solve}, \Psi_{solve} \rangle$ is a solution to $\Phi_{problem}$, and Φ_{solve} is satisfied by some state in which Cohen does have write access, then a corresponding behavior satisfying Ψ_{solve} must have the effect of revoking that access.

Productive problems can also be used to characterize scheduling problems. For example, to guarantee that executor x is not starved [Dijkstra 72], we must show that some history can be produced containing an operation executed by x. Formally, this situation can be characterized by the behavioral productive problem

$$\Psi_{problem}(\sigma, H) \equiv \delta \in H \wedge x = Executor(\delta)$$

In this chapter we have studied productive problems - constraints on behavior that are to be produced or locally produced. Productive problems and enforcement problems model a large number of problems encountered in computational systems. Even so, we find in the next chapter that all

protection problems (in particular, information problems) cannot be modelled in this way.

Chapter 7 - Problems and Solutions

"In this world, things are complicated and are decided by many factors. We should look at problems from different aspects, not just one."

Mao Tse Tung "On the Chungking Negotiations"

----- Section 7.1 --- Introduction

Throughout this thesis we have restricted our attention to behavioral problems, those that can be characterized as a constraint on behavior to be enforced or produced. We developed a model in which those problems could be solved by imposing a constraint and/or adding a mechanism to a given system. In this chapter, we expand our notion of problem to include any problem that may be solved by an initial constraint and a mechanism, developing a notation for a problem as a characteristic function of its solutions.

Certain problems may be described using this notation that cannot be appropriately described as behavioral problems. We find that enforcement problems cannot be used to model information problems, even though they appear able to at first. Solutions to enforcement problems are shown to satisfy two properties, the Containment property and the Join property, neither of which may be satisfied by solutions to certain information problems.

We explore the constraints (boundary conditions) one might wish to require of solutions to problems and consider ways of measuring and comparing solutions. Finally, we explore the relationships between maximal (least restrictive) solutions and those which are optimal according to some measure, paying special attention to a class of measures we define as monotonic.

----- Section 7.2 --- Characterizing Problems

We will formally characterize a problem as a predicate χ such that $\chi(\varphi,M)$ only if $\langle\varphi,M\rangle$ solves some given problem. For example, the enforcement problem $\Psi_{problem}$ would be characterized by

$$\chi(\varphi,M) \equiv \langle\varphi,M\rangle \text{ enforces } \Psi_{problem}$$

The productive problem $\Phi_{problem}$ is characterized by

$$\chi(\varphi,M) \equiv \langle\varphi,M\rangle \text{ produces } \Phi_{problem}$$

In cases where we do not consider adding any mechanism to a system, we will instead write χ as a predicate on an initial constraint alone. For example

$$\chi(\varphi) \equiv \varphi \text{ enforces } \Psi_{problem}$$

The χ notation makes it easy to describe and compare properties of solutions to problems. First, we develop some notation

>> Def 7-1] $\varphi 1 \wedge \varphi 2$, $\varphi 1 \vee \varphi 2$

$$\varphi 1 \wedge \varphi 2 \equiv_{def} \lambda\sigma.(\varphi 1(\sigma) \wedge \varphi 2(\sigma))$$
$$\varphi 1 \vee \varphi 2 \equiv_{def} \lambda\sigma.(\varphi 1(\sigma) \vee \varphi 2(\sigma))$$

Earlier, we defined (definition 2-15)

$$\varphi 1 \subseteq \varphi 2 \equiv_{def} (\forall\sigma)(\varphi 1(\sigma) \supset \varphi 2(\sigma))$$

We extend that definition so that

>> Def 7-2] $\langle\varphi 1,M1\rangle$ contained in $\langle\varphi 2,M2\rangle$

$$\langle\varphi 1,M1\rangle \subseteq \langle\varphi 2,M2\rangle \equiv_{def}$$
$$\{ \tau_{M1}(\sigma',H') \mid (M1:\varphi 1)(\sigma') \} \subseteq \{ \tau_{M2}(\sigma',H') \mid (M2:\varphi 2)(\sigma') \}$$

That is, $\langle\varphi 1,M1\rangle \subseteq \langle\varphi 2,M2\rangle$ if all behaviors that can occur when $\langle\varphi 1,M1\rangle$ is imposed on a system can also occur when $\langle\varphi 2,M2\rangle$ is imposed.

If $\Psi_{problem}$ is an enforcement problem, then any solution that enforces $\Psi_{problem}$ is contained in any solution that induces $\Psi_{problem}$, since the latter solution is guaranteed to permit all behaviors satisfying $\Psi_{problem}$. Formally

Theorem 7-1]

If $\langle\Phi 1, M1\rangle$ enforces $\Psi_{problem}$
and $\langle\Phi 2, M2\rangle$ induces $\Psi_{problem}$

then $\langle\Phi 1, M1\rangle \subseteq \langle\Phi 2, M2\rangle$

which follows directly from definitions 2-18 and 2-19.

Next we show that enforcement problems satisfy two important properties, the Containment property and the Join property.

Theorem 7-2] The Containment Property

$$\Phi 2 \subseteq \Phi 1 \supset . \quad \chi(\Phi 1) \supset \chi(\Phi 2)$$

where $\chi(\Phi) \equiv \Phi$ enforces $\Psi_{problem}$

That is, if $\Phi 1$ enforces $\Psi_{problem}$ and $\Phi 2$ is more restrictive than $\Phi 1$ (permitting a subset of the initial states permitted by $\Phi 1$), then $\Phi 2$ enforces $\Psi_{problem}$ as well.

Theorem 7-3] The Join Property

$$\chi(\Phi 1) \wedge \chi(\Phi 2) \supset . \quad \chi(\Phi 1 \vee \Phi 2)$$

where $\chi(\Phi) \equiv \Phi$ enforces $\Psi_{problem}$

That is, if $\Phi 1$ solves $\Psi_{problem}$ and $\Phi 2$ solves $\Psi_{problem}$, then a weaker solution will solve $\Psi_{problem}$ as well, one in which the initial states are constrained so that either $\Phi 1$ or $\Phi 2$ must be true.

More generally, we can show that

Theorem 7-4]

$$1) \quad \langle \varphi 2, M2 \rangle \subseteq \langle \varphi 1, M1 \rangle \supset. \quad \chi(\varphi 1, M1) \supset \chi(\varphi 2, M2)$$
$$2) \quad \chi(\varphi 1, M) \wedge \chi(\varphi 2, M) \supset. \quad \chi(\varphi 1 \vee \varphi 2, M)$$

where $\chi(\varphi, M) \equiv \langle \varphi, M \rangle$ enforces $\Psi_{problem}$

If we are to solve some problem χ without adding a mechanism, we may be interested in finding the maximal (least restrictive) solution to the problem. If the join of any two solutions is itself a solution, then the maximal solution is simply the join of all of the solutions.

$$\varphi_{max} \equiv \vee \{ \varphi \mid \chi(\varphi) \}$$

More generally, if χ does not satisfy the join property (for an example, see section 7.4), then we define a maximal solution as one that is no more restrictive than any other solution. Formally we define

>> Def 7-3] φ_{max} maximally solves χ iff

$$\chi(\varphi_{max}) \wedge$$
$$(\varphi_{max} \subseteq \varphi \wedge \chi(\varphi) \supset. \varphi_{max} = \varphi)$$

Theorem 7-5]

If $\chi(\varphi 1) \wedge \chi(\varphi 2) \supset. \chi(\varphi 1 \vee \varphi 2)$
and φ_{max} maximally solves χ

then $\varphi_{max} = \vee \{ \varphi \mid \chi(\varphi) \}$

----- Section 7.3 --- Constraining and Measuring Solutions

In section 4.2 we noted that the optimal solution to an enforcement problem $\Psi_{problem}$ need not be one that induces $\Psi_{problem}$, and in fact in section 4.3 we showed that if we were not permitted to add a mechanism in solving a problem (i.e. only impose an initial constraint), that no solution inducing $\Psi_{problem}$ might even be possible. In this section, we will

consider measures for solutions, including a class of measures, those that are monotonic, for which an optimal solution does correspond to one that induces $\Psi_{problem}$. More generally, we will consider the relationship between optimal and maximal (least restrictive) solutions.

Even if we are not concerned with optimality, we may nonetheless want to eliminate certain solutions from consideration. For example, any enforcement problem can be solved by "pulling the plug" - that is, by adding the mechanism M_{null} which never executes any base system operation, or by imposing the constraint $\Phi_{solve} \equiv ff$, which does not permit the system to operate in any initial state.

We restrict the class of acceptable solutions by requiring that solutions satisfy some constraint, which we write as $\chi_{constraint}$. For example, if we wished to guarantee that an initial constraint satisfying some enforcement problem be satisfied by some state in the set S, we would define

$$\chi(\Phi) \equiv \chi_{constraint}(\Phi) \wedge \Phi \text{ enforces } \Psi_{problem}$$

$$\underline{\text{where}} \quad \chi_{constraint}(\Phi) \equiv (\exists \sigma \in S)(\Phi(\sigma))$$

Suppose that we wanted to guarantee that for any solution $\langle \Phi, M \rangle$ to the enforcement problem $\Psi_{problem}$, the mechanism permitted the execution of at least one acceptable behavior from any state satisfying Φ. We would define

$$\chi(\Phi, M) \equiv \chi_{constraint}(\Phi, M) \wedge \langle \Phi, M \rangle \text{ enforces } \Psi_{problem}$$

$$\underline{\text{where}} \quad \chi_{constraint}(\Phi, M) \equiv \langle \Phi, M \rangle \text{ produces } \Psi_{problem}$$

As a final example, suppose that we want to guarantee that the solution to some enforcement problem $\Psi_{problem}$ can itself be locally produced by some set of users X in context $\Phi_{context}$. We define

$$\chi(\Phi, M) \equiv \chi_{constraint}(\Phi, M) \wedge \langle \Phi, M \rangle \text{ enforces } \Psi_{problem}$$

$$\underline{\text{where}} \quad \chi_{constraint}(\Phi, M) \equiv$$
$$(\exists M^*)(\langle \Phi_{context}, M^* \rangle \text{ locally produces } \langle \Phi, M \rangle \text{ for } X)$$

In the bank robbery problem (section 6.4), we found that $\Psi_{problem}$ could be enforced both by

$$\varphi_{solve}(\sigma) \equiv locked(\sigma.vault) \quad \text{and}$$
$$\varphi_{solve}(\sigma) \equiv drunk(\sigma.robber)$$

However, we noted that for some reasonable context, the local production constraint, formalized just above, eliminates the second of those solutions.

In the bank robbery example, the added constraint eliminated one possible solution. In fact, an added constraint may eliminate <u>all</u> solutions, except by <u>requiring</u> the addition of a mechanism. For example, suppose we conjoined with $\chi_{constraint}$ above, the requirement that

$$\varphi(\sigma) \supset. \neg locked(\sigma.vault) \wedge \sigma.vault.money \neq 0$$

The problem may still be solved, but not without adding a mechanism that can provide additional protection for the vault, since the added constraint requires that it not be locked. We add a mechanism that provides merciless killer dogs, such that the robber can only get to the vault if the dogs are not guarding it. That is, in place of the command $\delta 1$, the mechanism provides $\delta 1'$, defined so that

$$\tau_M(\delta 1')(\sigma') = \begin{array}{ll} \delta 1 & \underline{if} \quad \neg guarding(\sigma'.vault) \\ \lambda & \underline{otherwise} \end{array}$$

where $guarding(\sigma'.vault)$ indicates that the dogs are guarding the vault in state σ'. Addition of this mechanism solves the bank robbery problem so long as it is constrained so that

$$\varphi_M(\sigma') \equiv guarding(\sigma'.vault)$$

A likely constraint is that a solution be at least as good (according to some criteria) as a given solution. That is

$$\chi_{constraint}(\varphi,M) \equiv Worth(\varphi_{given},M_{given}) \leq Worth(\varphi,M)$$

where Worth is some measure of a solution's worth and "$\leq$" is some partial order of those worths.

Just as we wrote $\chi(\varphi)$ when no mechanism can be added, we write $Worth(\varphi)$ for the worth of a solution that does not admit a mechanism.

It is often reasonable to believe that if one solution permits all of the behavior permitted by another solution, it should be at least as worthy. Measures of worth that exhibit this property are monotonic.

>> Def 7-4] $\langle Worth, \leq \rangle$ is a monotonic measure iff

$$\langle \varphi 1, M1 \rangle \subseteq \langle \varphi 2, M2 \rangle \supset . \; Worth(\varphi 1, M1) \leq Worth(\varphi 2, M2)$$

We might value worth directly in terms of the behavior permitted, assuring a monotonic measure. Define

$$Worth(\varphi, M) \equiv \{ \tau_M \langle \sigma', H' \rangle \mid (M:\varphi)(\sigma') \}$$

with "$\leq$" defined to be set inclusion ($\subseteq$). Then one solution is more worthy than another if it permits more behavior. Yet this measure may be too restrictive. Consider the system

$\delta 1$: if $\alpha = 1$ then $\beta 1 \leftarrow m$
$\delta 2$: if $\alpha = 2$ then $\beta 1 \leftarrow m$
$\delta 3$: if $\alpha \geq 3$ then $\beta 2 \leftarrow m$

The problem

$$(markov)\ \Psi_{problem}(\sigma, \delta) \equiv \delta(\sigma).\beta 2 = \sigma.\beta 2$$

can be solved both by $\varphi 1$ and $\varphi 2$ where

$\varphi 1(\sigma) \equiv \sigma.\alpha \leq 1$
$\varphi 2(\sigma) \equiv \sigma.\alpha \leq 2$

$\varphi 2$ strictly contains $\varphi 1$, and therefore, by the measure above is more worthy. However, from a semantic viepoint, it may be argued that $\varphi 1$ is just as good as $\varphi 2$. Both solutions have the effect that the only behaviors permitted by the system transmit information from m to $\beta 1$ - no more and no less. It is clear that additional research on valid measures is necessary.

In [Cohen 76], we study a monotonic measure based on an information transmission formalism. Using that measure, one solution is as worthy as another if it permits at least the same information transmission paths.

Given a measure of the worth of a solution, we may naturally be interested in finding optimal solutions, those with maximal worths. Given some measure of worth, we may define (with respect to that measure)

>> Def 7-5] $\langle \Phi, M \rangle$ optimally solves X iff

$$Worth(\Phi, M) = Max \{ Worth(\Phi\star, M\star) \mid X(\Phi\star, M\star) \}$$

If X respresents an enforcement problem $\Psi_{problem}$, then any solution that induces $\Psi_{problem}$ is optimal, with respect to any monotonic measure. Formally

Theorem 7-6]

If $\langle \Phi, M \rangle$ induces $\Psi_{problem}$

then $\langle \Phi, M \rangle$ optimally solves X
with respect to any monotonic measure
where $X(\Phi, M) \equiv \langle \Phi, M \rangle$ enforces $\Psi_{problem}$

In particular, this theorem shows that an optimal solution may be found for any monotonic enforcement problem (with respect to a monotonic measure). In section 3.7, we showed how such a solution might be constructed.

If we are not permitted to add a mechanism to a system in order to solve an enforcement problem, then the optimal solution may not induce $\Psi_{problem}$ (see section 4.3). However, for any problem satisfying the Join property, a maximal solution is an optimal solution (for a monotonic measure). Formally

Theorem 7-7]

If $X(\varphi 1) \wedge X(\varphi 2) \supset X(\varphi 1 \vee \varphi 2)$
and φ_{max} maximally solves X

then φ_{max} optimally solves X
with respect to any monotonic measure

----- Section 7.4 --- Information Problems

In this section, we briefly discuss information problems, those concerned with guaranteeing that transmission of information from certain objects to other objects is prevented. We will show that, while it is tempting to treat information problems as enforcement problems, solutions to information problems do not satisfy the properties that must be satisfied by solutions to enforcement problems.

We could write (adapted from [Denning 75])

$$\alpha\text{-}(\sigma:H)\text{->}\beta$$

to mean that when **H** is executed in state σ, information can flow from object α to object β.

Suppose that we wanted to solve the problem: prevent transmission of information from α to β. This can be represented by the enforcement problem

$$\Psi_{problem}(\sigma, H) \equiv \neg\, \alpha\text{-}(\sigma:H)\text{->}\beta$$

that is, acceptable behaviors are those in which no information can flow from α to β. The problem can be solved by φ where

φ enforces $\Psi_{problem}$

We will show that this formalism for information problems is incorrect, for the solutions to the problem of preventing information transmission from α to β do not correspond to the solutions to $\Psi_{problem}$, and in fact, the solution cannot correspond to the solutions of any enforcement problem.

Consider the system

$\delta 1$: $\beta \leftarrow q$
$\delta 2$: <u>if</u> m <u>then</u> $\beta \leftarrow \alpha$

Transmission of information from α to β may be prevented by imposing the constraint

$$\varphi(\sigma) \equiv \neg\sigma.m$$

That is, if m is constrained to be false, execution of $\delta 2$ will not transmit information from α to β. Now according to theorem 7-2, if an information problem is a behavioral problem, a more restrictive solution should prevent information transmission from α to β as well. Consider the stricter constraint (the more restrictive solution)

$$\varphi(\sigma) \equiv \neg\sigma.m \wedge \sigma.q = \sigma.\alpha$$

In addition to requiring that m be false, φ requires that the *initial values* of q and α be the same. However, since α and q are the same, execution of $\delta 1$ transmits information about α to β, for an observer of β can (after $\delta 1$ has executed) infer the initial value of q, and knowing φ, can thereby infer the initial value of α! Solutions to information problems do <u>not</u> satisfy the Containment Property.

Next consider the system

δ: $\beta \leftarrow \alpha$

A solution that prevents transmission of information from α to β is

$$\varphi 1(\sigma) \equiv \sigma.\alpha = 23$$

If the value of α is initially constrained to be 23, no information can be transmitted from α to β. The amount of information transmitted from a source to a receiver depends upon the number of messages that can be transmitted and the probability of transmission of each one [Shannon & Weaver 49]. If only one message can be sent, no information can be

transmitted. By constraining α to be 23, only one message can be sent from α to β as the result of executing δ - the "message" 23. An equally good solution is

$$\varphi 2(\sigma) \equiv \sigma.\alpha = 96$$

However, $\varphi = \varphi 1 \vee \varphi 2$ is <u>not</u> a solution. If α may initially be either 23 or 96, then two messages can be sent from α to β, and information can be transmitted from α to β. Therefore, solutions to information problems do not satisfy the Join Property (theorem 7-3) either.

Solutions to information problems may not satisfy the Containment Property or the Join Property because of the inferential nature of initial constraints. If the value of an object is constrained to be constant, no inference can be made about the object, and therefore no information can be transmitted from it. If the value of an object is constrained to correspond to the value of another object, an inference may be made about that other object, and information may be transmitted from it.

In effect, the initial constraint itself partially determines which behaviors are acceptable. This suggests that a formal theory of information transmission must consider the constraint initially placed on a system. [Millen 76] does note the importance of considering constraints in analyzing information paths in sequential programs. A formalism for studying information transmission in the presence of constraint is developed in [Cohen 76].

Chapter 8 - Conclusion

This thesis has developed formalisms for the concepts - problem, mechanism and solution - useful for proving properties about real systems. In this conclusion, we will describe the important contributions of these formalisms and their value as part of a continuing study of computational systems.

These formalisms are based upon a simple description of computational systems as a state space and a set of discrete indivisible operations. Parallelism is not modelled directly, but can be modelled through the addition of a mechanism that interleaves the operations in appropriate ways.

States are described as being wholly comprised of objects having fixed unique names. As a result, it is possible to designate objects as belonging to some class (e.g. "trusty" or "sensitive" as in chapter 5). The executor of (process executing) an operation corresponds to the name of some object. That name is part of the name of the operation and can be determined independently of the state in which the operation executes. These aspects of the model were responsible for simplifying descriptions of problems and proofs of the correctness of their solutions.

We defined a mechanism so that it can be used uniformly to model those things commonly called "mechanisms", including protection mechanisms, sequential and parallel control mechanisms, synchronization mechanisms, levels of hierarchy and interpreters, but commonly described in diverse ways. We categorized mechanisms in terms of algebraic properties, describing them as direct, homomorphic, markov and consistent. These properties correspond to properties one might ordinarily consider in studying these mechanisms. The latter two properties, for example, relate to concurrency.

We did not consider properties of mechanisms beyond those required for the purposes of the thesis. Mechanisms do seem to have a rich algebraic structure; they are related to or extensions of structures found in automata theory, formal language theory and category theory. We also did not study embedding of mechanisms (e.g. - the synchronization mechanism embedded in the multiprogram control mechanism - section 3.5). These may be useful research topics.

We described mechanisms in terms of an elegant notation that aided the description and development of a number of proof techniques. Proof techniques known to be useful in studying one mechanism may be cast in a more general framework and applied in studying a different mechanism. For example, the method of static invariants [Robinson & Holt 74] used in studying synchronization was shown here to be applicable to protection.

Systems may contain multiple mechanisms, each of which may need to be constrained in order to solve some problem. A uniform mechanism formalism permits the development of a formal methodology for determining those joint constraints. That methodology was developed in section 4.4. It was used in chapter 5 to find a solution to a protection problem that specified both an initial constraint on access rights as well as a property that had to be satisfied by the programs to be executed by certain users.

This thesis primarily has studied a class of problems we called behavioral problems, those that can be described as a constraint on the behavior of a system. Using behavioral specifications, both protection and synchronization problems can be described independently of any system in which that problem might be solved. This means that we can show, very naturally, how a protection mechanism might, for example, be used to solve a synchronization problem, as when mutual exclusion is enforced by a protection mechanism that permits non-sharable capabilities (i.e. only one process at a time has access rights for some object).

We showed how the behavior of a system could be constrained by adding a mechanism to it or by imposing some constraint on the states in which the system might initially be permitted to operate. This led us to define a solution as a < constraint, mechanism > pair. We described a number of primitive relationships between < constraint, mechanism > pairs and behavioral constraints, including induce, enforce, produce and locally produce. These descriptions can be used in various ways as part of a language for problem specification. For example, "enforce" is used in describing that class of problems (including many synchronization and protection problems) solved by guaranteeing that execution of only certain acceptable behaviors are to be permitted. "Produce" is used in describing that class of problems (including other protection problems such as the revocation problem, and scheduling problems) solved by guaranteeing that at

least one acceptable behavior may always be executed. The primitives can be combined in various ways to describe numerous sorts of problems.

Behavioral problem specifications can be independent of any particular system, and therefore quite general. As a result, they may not be in a form well suited for proving the correctness of a solution in some given system. When a particular system is given, it may be possible to convert the behavioral specification into a static one - specified as a property of the state of the system. It is often easier to prove that some property of a system is invariant than to prove that only acceptable behaviors can be executed. We formally demonstrated the validity of certain conversions from behavioral to static specifications and also developed techniques for proving invariance. This technique was used successfully in chapter 5.

We discussed measures for solutions and characterized a property of measures - monotonicity - which requires that the worthiness of a solution increase as its generality increases. We did not consider particular measures in great detail. We might have liked to present a measure that would have formally indicated how the three solutions for the problem discussed in chapter 5 differed. Future research might explore useful measures and might consider ways of proving optimality of solutions beyond those considered in chapter 7.

Finally, we briefly discussed information problems, those concerned with preventing certain information transmission in computational systems. We showed that one might think of information problem as enforcement problems - an acceptable behavior being one in which illegal information flow does not take place. However, we showed that in general, information problems cannot be described in this way, for their solutions satisfy neither the Containment property nor the Join property, which we proved had to be satified by solutions to enforcement problems.

In [Cohen 76] we continue the study of information problems, presenting a definition of information transmission based on ideas taken from information theory, and using those definitions to formally define a simple version of the Confinement problem and the Military Security problem. A study of a more complex version of the Confinement Problem (including "declassification") is in progress, including proof of a solution in a system

similar to that described in Appendix A, modified to correspond to the mechanism provided to solve Confinement in the Hydra system [Cohen & Jefferson 75].

Mechanisms may be added to a system to prevent information transmission. However, [Rotenberg 73] and [Denning 75] have shown that mechanisms may subtly add new paths for information transmission. Research in progress indicates that the formal definition of mechanism developed in this thesis may be useful in understanding how the addition of those information paths may be prevented. In particular, strongly consistent mechanisms do not add information transmission paths. Since many of the implementation levels of an operating system may be viewed as consistent mechanisms, the mechanism formalism may be useful in simplifying proofs of the information security of an operating system (e.g. [Millen 76]).

Finally, while this thesis has provided a framework for describing protection problems, it has not discussed any specification language for them, in the sense that path expressions [Campbell & Habermann 75] are a specification language for synchronization problems. Certainly, the relations induce, enforce, produce and locally produce should be included (or derivable from other constructs) in any such language.

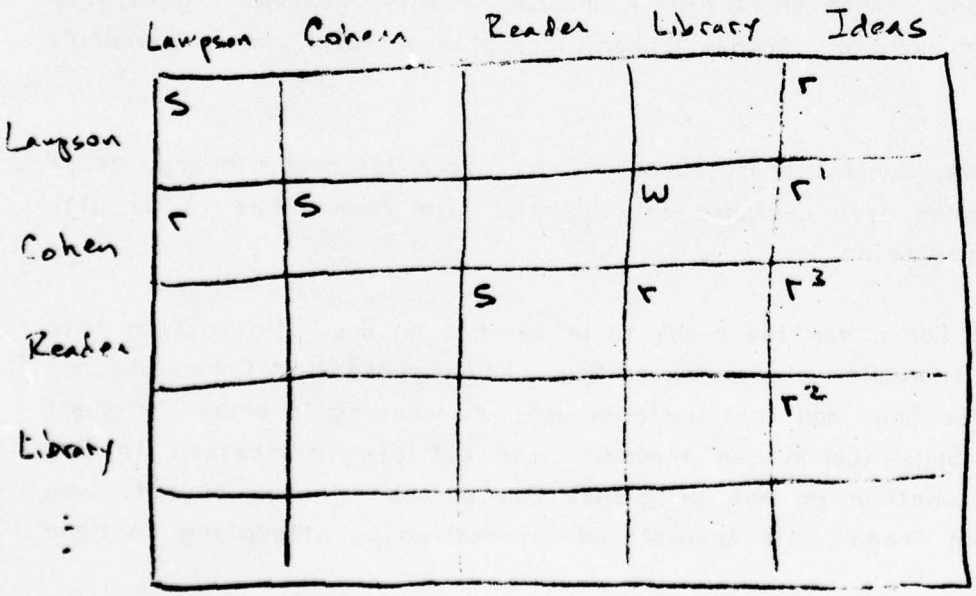
Appendix A - Access Matrix Systems

In this appendix, we will describe Access Matrix systems and present a simple example of one that will be used in chapter 5.

[Lampson 71] was the first to describe protection in terms of an access matrix, A, where A[x,y] contains the set of rights x has for y. For example, if r represents the right to read, and w represents the right to write, then Cohen will be allowed to read the Salary File only if $r \in A[Cohen, Salary]$ and Cohen will be able to write the Salary file only if $w \in A[Cohen, Salary]$.

We will use the notation <x,y> as a shorthand for A[x,y].

We will assume a base system which provides four generic operations. We delay our discussion of two these, call and create. The operation move(x,y,j,z,k,n), when executed by x, moves a portion of the contents of z to y. The access matrix mechanism only allows this operation to execute if x has the right to read z and to write y. The operation $op_\alpha(x,y,i,j,k)$ when executed by x, performs some operation on the contents of y. The mechanism only permits this operation to execute if x has both the right to read and write y. In general, an access matrix mechanism tests a conjunction of conditions, each of the form, $q \in \langle x,y \rangle$, in order to determine whether some operation should be permitted to execute.



(superscripts referred to in text)

Since the mechanism contains a mechanism state (the matrix), there are operations that manipulate this state. For example, the operation (1) $take_r(x,y,z)$ takes a copy of y's r-right for z and gives it to x. Thus, $take_r$(Cohen,Lampson,Ideas) gives Cohen read access to Lampson's Ideas. There are two pre-requisites for this operation. Lampson must have r-rights for his Ideas and Cohen must have r-rights for (be able to read) Lampson.

The "grant" operation is the complement of the "take" operation. Once Cohen has taken r-rights for Lampson's Ideas, Cohen can execute (2) $grant_r$(Cohen,Library,Ideas) which grants Library the right to read (r-rights for) Ideas. Subsequently, Reader might execute (3) $take_r$(Reader,Library,Ideas) to get r-rights for Ideas, so that Reader's attempts to use Ideas will be successful (not be prevented by the mechanism).

In Lampson's description of the access matrix, the matrix was not square. The rows only listed "subjects", those objects that represented executors. The matrix we describe here is square so that passive entities like Library can have the right to access objects, if only so subjects can "take" these rights. As a result, we need some formal way to distinguish subjects from other objects. We specify a new right, s-rights, so that x can be an executor only if $s \in \langle x,x \rangle$.

As "grant" and "take" share rights around, "remove" removes rights from the matrix. For example, $remove_r$(Cohen,Library,Ideas) will remove Library's right to Ideas.

Both $remove_r$(Cohen,Library,Ideas) and $grant_r$(Cohen,Library,Ideas) require that Cohen have w-rights for Library. The reason has to do with information transmission.

Suppose that Cohen has the right to write into an Spy. By writing into Spy, Cohen can transmit information to Spy. Now suppose that Cohen did not have w-rights for Spy, and that w-rights were unnecessary in order to grant some right to Spy. Cohen can transmit (one bit of) information to Spy depending upon whether or not he grants Spy access to some object, say Salary. Spy can "read" this transmitted information by attempting to read

(via a "move") Salary. If Spy succeeds, Cohen has transmitted a "1"; If Spy fails, Cohen has transmitted a "0".

Requiring w-rights for "grant"s (and "remove"s), as well as for for "writes" ("move"s and "op"s), lets us consistently treat w-rights as an indication of whether or not information can be transmitted. A discussion of these and related issues can be found in [Cohen & Jefferson 75].

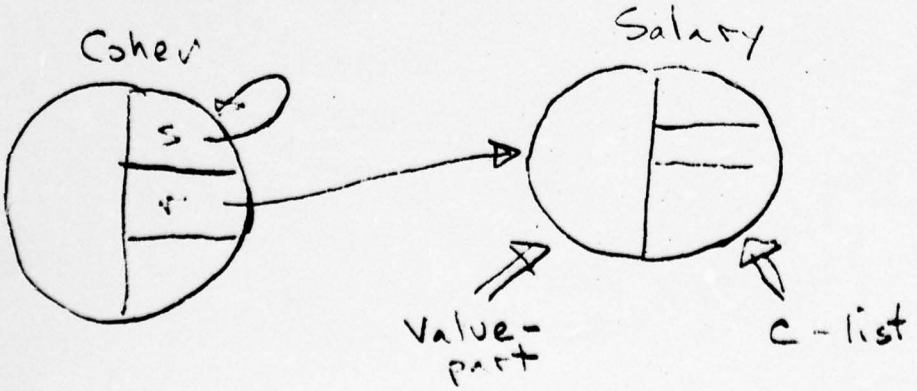
The example system contains a "create" operation that creates a new object, and gives the executor of the operation all rights for that new object. To simplify the formal treatment of objects, we will assume that a new object is not actually created. The system provides the name of an object not yet used - one for which no other object has access rights.

The protection state (the protection matrix) could be modelled as bundled up in a single object A. Then $\sigma.A[x,y]$ (also written as $\langle x,y\rangle(\sigma)$) would contain the set of rights x had for y in state σ.

It is more useful to model the protection matrix as a Capability system (this system is in fact, a simplification of the HYDRA system [Wulf 74]). We consider all of x's rights for other objects to be part of x itself. We divide each object into two components, its "actual" contents, called its Value-part and its set of rights, called its C-list (for capability list - the set of rights that x has for y is called x's capability for y). In this model, we define

$$\langle x,y\rangle(\sigma) \equiv_{def} \sigma.x.C\text{-}list[y]$$

In the diagram below, Cohen is a subject with the right to read the Salary file.



This treatment of the protection state is especially useful in analyzing information transmission. Information can be transmitted to some object if that object can be written into. As we noted above, x can also transmit information to y if x can change the rights y has for other objects. If y's rights for other objects are part of y itself, then manipulation of the protection state need not be treated as a special case in analyzing information transmission.

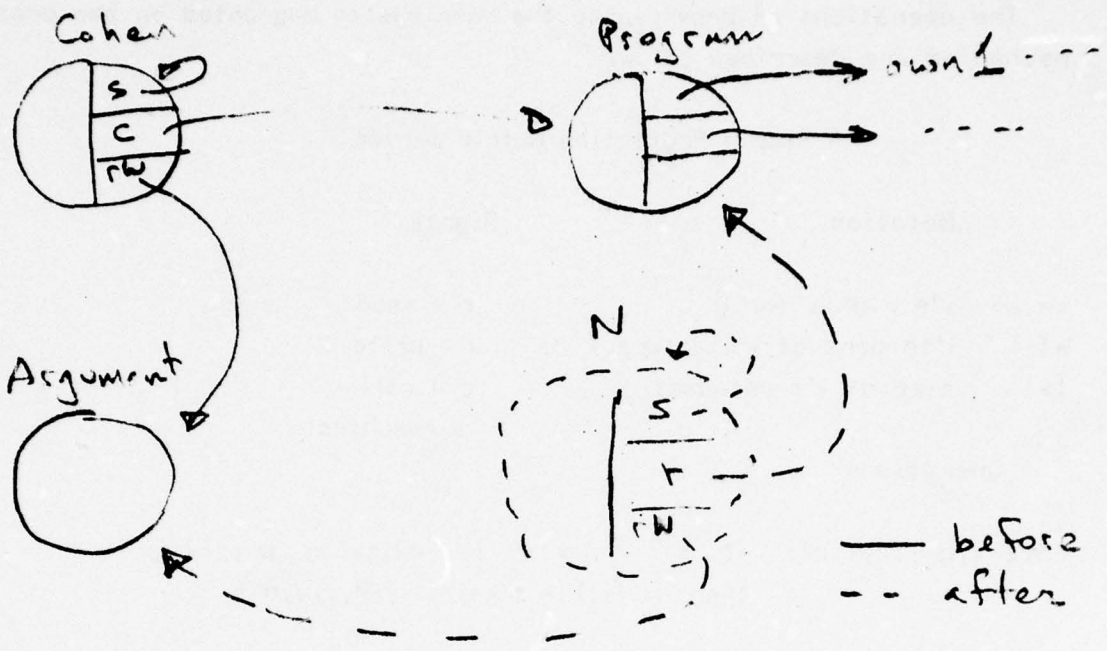
A thesis by [Rotenberg 73] presents a collection of examples that show how subtle (and not so subtle) manipulations of the protection state can be used to transmit arbitrary amounts of information. By representing the protection state explicitly in the manner described above, these covert information channels can be detected directly using the methods of analysis described in [Cohen 76].

Now we can finally describe the "call" operation. "Call" is a simple-minded version of a re-entrant subroutine call (with no corresponding return). Programs (subroutines) have two distinct parts.

1. The code that will execute. In our system, this is assumed to be in the Value-part of the object representing the program.

2. The set of objects every incarnation of the program will need to access (i.e. own variables rather than arguments). The C-list of the program object contains capabilities for these objects.

The diagram below depicts an example in which Cohen calls a Program, passing along some Argument, call(Cohen,Program,Argument) (the dotted dotted lines represent "after the call").



When Program is called, a new object N is created. N gets a capability containing the same rights Cohen had to Argument so that N can access Argument. It also gets r-rights to Program, so that by executing "take"s, it can get access to any of the objects in Program's C-list.

The Value-part of N is an exact copy of the Value-part of Program. This insures N will execute the appropriate code (see section 3.5).

In order to call Program, Cohen must have c-rights for Program. In this way, Cohen may be able to call a program without necessarily being able to inspect or alter Program's code (Value-part) or gain access (via "take") to the objects that Program (that is, each of its incarnations) can access. The creation of a new object (incaranation) N of the program for each "call" not only allows re-entrancy, but it is also important for solving a number of classic protection problems (e.g. Mutual Suspicion)

Note that no object, including the caller, gets any right to access the incarnation, N, of the program. This means that (unlike other examples of access matrix system, [Lampson 71], [Graham & Denning 72]), subjects cannot control each other directly, but can only communicate by transmitting information through intermediate objects.

The operations as provided by the base system augmented by the protection mechanism are described below.

A Simple Protection Matrix System

Notation		Rights
<x,y>	x's rights for y	r - read
x[i]	i'th word of x's datapart	w - write
|x|	size of x's datapart	c - call
		s - subject

Operations

move(x,y,j,z,k,n): if $s \in \langle x,x\rangle \wedge r \in \langle x,z\rangle \wedge w \in \langle x,y\rangle$
then $y[j+i] \leftarrow z[k+i]$, $i=0,\ldots,n-1$

$op_\alpha(x,y,i,j,k)$: if $s \in \langle x,x\rangle \wedge \{r,w\} \subseteq \langle x,y\rangle$
$\alpha \in \{add,\ldots\}$ then $y[i] \leftarrow \alpha(y[j],y[k])$

$take_\alpha(x,y,z)$: if $s \in \langle x,x\rangle \wedge r \in \langle x,y\rangle \wedge \alpha \in \langle y,z\rangle$
$\alpha \in \{r,w,c\}$ then $\langle x,z\rangle \leftarrow \langle x,z\rangle \cup \{\alpha\}$

$grant_\alpha(x,y,z)$: if $s \in \langle x,x\rangle \wedge w \in \langle x,y\rangle \wedge \alpha \in \langle x,z\rangle$
$\alpha \in \{r,w,c\}$ then $\langle y,z\rangle \leftarrow \langle y,z\rangle \cup \{\alpha\}$

$remove_\alpha(x,y,z)$: if $s \in \langle x,x\rangle \wedge w \in \langle x,y\rangle$
$\alpha \in \{r,w,c\}$ then $\langle y,z\rangle \leftarrow \langle y,z\rangle - \{\alpha\}$

create(x): if $s \in \langle x,x\rangle$
then let N be new object in
$\langle x,N\rangle \leftarrow \{r,w,c\}$

call(x,y,z): if $s \in \langle x,x\rangle \wedge c \in \langle x,y\rangle$
then let N be new object in (
$\langle N,N\rangle \leftarrow$ [redacted] $\{s\}$;
$N[i] \leftarrow y[i]$, $i = 1,\ldots,|y|$;
$\langle N,y\rangle \leftarrow \{r\}$;
$\langle N,z\rangle \leftarrow \langle x,z\rangle$)

Appendix B - Proofs

Theorem 2-1

Given:

1] τ is markov

Prove: $\tau(H1'H2')(\sigma') = \tau(H1')(\sigma') \,\&\, \tau(H2')(H1'(\sigma'))$
by induction on length of H2'
Base H2' = λ. Direct by subsitution and def 2-8
Induction Assume for H2', prove for H2'δ'

$\tau(H1'H2'\delta')(\sigma')$
= $\tau(H1'H2')(\sigma') \,\&\, \tau(\delta')(\,(H1'H2')(\sigma')\,)$ [1]
= $\tau(H1')(\sigma') \,\&\, \tau(H2')(H1'(\sigma')) \,\&\, \tau(\delta')((H1'H2')(\sigma'))$ [Induction]
= $\tau(H1')(\sigma') \,\&\, \tau(H2')(H1'(\sigma')) \,\&\, \tau(\delta')(H2'(H1'(\sigma')))$
= $\tau(H1')(\sigma') \,\&\, \tau(H2'\delta')(H1'(\sigma'))$ [1]

Theorem 2-2

Given:

1] M is markov

2] $\tau_M(\,\delta'(\sigma')\,) = (\,\tau_M(\delta)(\sigma')\,)(\,\tau_M(\sigma')\,)$

Prove: M is homomorphic
that is: $\varphi_M(\sigma') \supset \tau_M(\,H'(\sigma')\,) = (\,\tau_M(H')(\sigma')\,)(\,\tau_M(\sigma')\,)$
We will prove the more general result:
$\tau_M(\,H'(\sigma')\,) = (\,\tau_M(H')(\sigma')\,)(\,\tau_M(\sigma')\,)$
by induction on length of H'
Base H' = λ. Direct by substitution and def 2-8
Induction Assume for H', prove for H'δ'

$\tau_M((H'\delta')(\sigma'))$
= $\tau_M(\delta'(H'(\sigma')))$
= $(\,\tau_M(\delta')(H'(\sigma'))\,)(\,\tau_M(H'(\sigma'))\,)$ [2]
= $(\,\tau_M(\delta')(H'(\sigma'))\,)(\,(\,\tau_M(H')(\sigma')\,)(\,\tau_M(\sigma')\,)\,)$ [Induction]
= $(\,\tau_M(H')(\sigma') \,\&\, \tau_M(\delta')(H'(\sigma'))\,)(\,\tau_M(\sigma')\,)$
= $(\,\tau_M(H'\delta')(\sigma')\,)(\,\tau_M(\sigma')\,)$ [1]

Theorem 3-1

Given:

1] τ_M is state isomorphic

Prove: Φ_M = tt
Proof by contradiction

2] Assume $\neg\Phi_M(\sigma 1')$
3] Let $\tau_M(\sigma 1') = \sigma$
4] $\sigma \in |\,\tau_M(\sigma')\,|$ [3]
5] $\sigma \in |\,\tau_M(\sigma')\,|\,\Phi_M(\sigma')\,|$ [4, Def of M]
6] $(\exists\sigma 2' \neq \sigma 1')(\,\tau_M(\sigma 2') = \sigma\,)$ [2,3,5]
Q.E.D. [3,6 contradicts 1]

Lemma A-1

Given:
1] τ is direct
2] τ is markov
3] $\tau(H')(\sigma') = H\delta$

Prove: $(\exists H_{\alpha} \leq H')(\ \tau(H_{\alpha})(\sigma') = H\ )$
by induction on length of H'
Base $H' = \delta'$

3] Assume $\tau(\delta')(\sigma') = H\delta$
4] $H = \lambda$ [1,3]
5] $\tau(\lambda)(\sigma') = \lambda$ [2]
6] $\lambda \leq \delta'$
Q.E.D. [3-(5,6)]

Induction Assume for H', prove for $H'\delta'$

7] Assume $\tau(H'\delta')(\sigma') = H\delta$
8] $\tau(H')(\sigma') \ \&\ \tau(\delta')(H'(\sigma')) = H\delta$ [7,2]
9] $\tau(\delta')(H'(\sigma')) = \delta \ \vee\ \tau(\delta')(H'(\sigma')) = \lambda$ [8,1]
10] Case 1 $\tau(\delta')(H'(\sigma')) = \delta$
11] $\tau(H')(\sigma') = H$ [8,10]
12] $H' \leq H'\delta'$
13] Case 2 $\tau(\delta')(H'(\sigma')) = \lambda$
14] $\tau(H')(\sigma') = H\delta$ [8,13]
15] $(\exists H_{\alpha} \leq H')(\ \tau(H_{\alpha})(\sigma') = H\ )$ [14, Induction]
16] $(\exists H_{\alpha} \leq H'\delta')(\ \tau(H_{\alpha})(\sigma') = H\ )$ [15]
Q.E.D. [9,10-(11,12),13-16]

Theorem 3-2

Given:
1] M is a runtime mechanism
2] τ_M is state isomorphic
3] M induces Ψ

Prove: Ψ is markov
That is: $\Psi(\sigma, H\delta) = \Psi(\sigma, H) \wedge \Psi(H(\sigma), \delta)$

4] Case 1 $\neg\Psi(\sigma, H)$
5] $\neg\Psi(\sigma, H\delta)$ [1,3,4, Th 3-4]

6] Case 2 $\Psi(\sigma, H)$

7] ==> Assume $\Psi(\sigma, H\delta)$
8] $(\exists\sigma', H')(\ \tau_M\langle\sigma', H'\rangle = \langle\sigma, H\delta\rangle\ )$ [3,7]
9] $\tau_M(\sigma') = \sigma$ [8]
10] $\tau_M(H')(\sigma') = H\delta$ [9]
11] $(\exists H1'H2' = H')(\ \tau_M(H1')(\sigma') = H\ )$ [10, Lemma A-1]
12] $\tau_M(H2')(H1'(\sigma')) = \delta$ [10,11, (2, Th 2-1)]
13] $\tau_M(H1')(\sigma') = H$ [11]
14] $\tau_M(H1'(\sigma')) = H(\sigma)$ [13,9, 1(τ_M homomorphic), 2(Th 3-1)]
15] $\tau_M\langle H1'(\sigma'), H2'\rangle = \langle H(\sigma), \delta\rangle$ [12,14]
16] $\Psi(H(\sigma), \delta)$ [15,3]

17] <== Assume $\Psi(H(\sigma), \delta)$
18] $(\exists\sigma', H')(\ \tau_M\langle\sigma', H'\rangle = \langle\sigma, H\rangle\ )$ [6,3]
19] $\tau_M(\sigma') = \sigma$ [18]
20] $\tau_M(H')(\sigma') = H$ [18]
21] $\tau_M(H'(\sigma')) = H(\sigma)$ [19,20, 1(τ_M homomorphic)]
22] $(\exists\sigma_{\alpha}, H_{\alpha})(\ \tau_M\langle\sigma_{\alpha}, H_{\alpha}\rangle = \langle H(\sigma), \delta\rangle\ )$ [17,3]
23] $\sigma_{\alpha} = H'(\sigma')$ [21,22,2]
24] $\tau_M(H_{\alpha})(H'(\sigma')) = \delta$ [22,23]
25] $\tau_M(H'H_{\alpha})(\sigma') = H\delta$ [20,24, 1(τ_M markov)]
26] $\tau_M\langle\sigma', H'H_{\alpha}\rangle = \langle\sigma, H\delta\rangle$ [19,25]
27] $\Psi(\sigma, H\delta)$ [26,3]

Theorem 3-3

Given:
1] Ψ is markov

Prove: Ψ is monotonic
That is: $\Psi(\sigma, H \& H^*) \supset \Psi(\sigma, H)$
by induction on length of H^*
Base $H^* = \lambda$. Direct by substitution
Induction Assume for H^*, prove for $H^*\delta$

2] Assume $\Psi(\sigma, H \& H^*\delta)$
3] $\Psi(\sigma, H \& H^*)$ [2,1]
4] $\Psi(\sigma, H)$ [3, Induction]

Theorem 3-4

Given:
1] M is a runtime mechanism
2] M induces Ψ

Prove: Ψ is monotonic
That is: $\Psi(\sigma, H1 \& H2) \supset \Psi(\sigma, H1)$
by induction on length of H2
Base $H2 = \lambda$. Direct by substitution.
Induction Assume for H2, prove for H2δ

3] Assume $\Psi(\sigma, H1H2\delta)$
4] $(\exists\sigma', H')(\tau_M\langle\sigma', H'\rangle = \langle\sigma, H1H2\delta\rangle$ [2,3]
5] $\tau_M(\sigma') = \sigma$ [4]
6] $\tau_M(H')(\sigma') = H1H2\delta$ [4]
7] $(\exists H^* \leq H')(\tau_M(H^*)(\sigma') = H1H2)$ [6,1, Lemma A-1]
8] $\tau_M\langle\sigma', H^*\rangle = \langle\sigma, H1H2\rangle$ [5,7]
9] $\Psi(\sigma, H1H2)$ [8,2]
10] $\Psi(\sigma, H1)$ [9, Induction]

The following sequence of Lemmas are preparation for the proof of theorems 3-5 and 3-6, that is, M_Ψ is a decision mechanism, and if Ψ is monotonic, then there is a Φ such that $\langle\Phi, M_\Psi\rangle$ induces Ψ.

We define M_Ψ in section 3.7 so that for each operation δ defined in the base system, M_Ψ provides a corresponding operation δ', and we denote this correspondence by writing $\delta' \sim \delta$. We will find it convenient to extend this correspondence to histories so that if, for example, $\delta1' \sim \delta1$ and $\delta2' \sim \delta2$, then we could write $\delta1'\delta2'\delta1' \sim \delta1\delta2\delta1$.

Lemma M-1

Given:
1] $\Phi_\Psi(\sigma')$
2] $\tau_\Psi(\sigma') = \sigma$

Prove: $(\forall H')(H'(\sigma').NM^* = \sigma)$
[note $M_\Psi = \langle\tau_\Psi, \Phi_\Psi\rangle$ defined in section 3.7]
by induction on length of H'
Base $H' = \lambda$

3] $\lambda(\sigma').NM^* = \sigma'.NM^*$
4] $\sigma'.NM^* = \sigma'.NM$ [1]
5] $\sigma'.NM = \sigma$ [2, def of τ_Ψ]

Induction Assume for H', prove for H'δ'

$(H'\delta')(\sigma').NM^*$
$= \delta'(H'(\sigma')).NM^*$
$= H'(\sigma').NM^*$ [def of δ']
$= \sigma$ [Induction]

Lemma M-2

Given:
1] $H' \sim H$
2] $\Phi_\Psi(\sigma')$
3] $\tau_\Psi(\sigma') = \sigma$

Prove: $\tau_\Psi(H')(\sigma') = H'(\sigma').HIST = H/_\sigma\Psi$
by induction on length of H'
Base $H' = \lambda$

4] $\tau_\Psi(\lambda)(\sigma') = \lambda$ [Def of τ_Ψ]
5] $\sigma'.HIST = \lambda$ [2]
6] $\lambda/_\sigma\Psi = \lambda$

Induction Assume for H', prove for $H'\delta'$

$(H'\delta')(\sigma').HIST$
$= \delta'(H'(\sigma')).HIST$
$=$ Let R be $H'(\sigma').HIST$ in
if $\Psi(H'(\sigma').NM\sigma, R\delta)$
then $R\delta$ else R
$=$ Let R be $H/_\sigma\Psi$ in [Induction]
if $\Psi(\sigma, R\delta)$ [Lemma M-1]
then $R\delta$ else R
$= (H\delta)/_\sigma\Psi$

$\tau_\Psi(H'\delta')(\sigma')$
$= \tau_\Psi(H')(\sigma') \;\&\; \tau_\Psi(\delta')(H'(\sigma'))$ [τ_Ψ markov]
$= H/_\sigma\Psi \;\&\; \tau_\Psi(\delta')(H'(\sigma'))$ [Induction]
$= H/_\sigma\Psi \;\&\; \delta$ when $\Psi(H'(\sigma').NM\sigma, H'(\sigma').HIST \;\&\; \delta)$
$H/_\sigma\Psi$ otherwise
$=$ Let R be $H/_\sigma\Psi$ in
$R\delta$ when $\Psi(\sigma, R\delta)$ [Lemma M-1, Induction]
R otherwise
$= (H\delta)/_\sigma\Psi$

Lemma M-3

Given:
1] $H' \sim H$
2] $\tau_\Psi(\sigma') = \sigma$
3] $\Phi_\Psi(\sigma')$

Prove: $(H/\Psi)(\sigma) = \tau_\Psi(H'(\sigma'))$
by induction on length of H'
Base $H' = \lambda$. Follows directly by [1,2]
Induction Assume for H', prove for $H'\delta'$

$\tau_\Psi((H'\delta')(\sigma'))$
$= \tau_\Psi(\delta'(H'(\sigma')))$
$= \delta'(H'(\sigma')).NM$
$= \delta(H'(\sigma').NM)$ when $\Psi(H'(\sigma').NM\sigma, H'(\sigma').HIST \;\&\; \delta)$
$H'(\sigma').NM$ otherwise
$=$ Let R be $H/_\sigma\Psi$ in
$\delta(R(\sigma))$ when $\Psi(\sigma, R\delta)$ [Lemmas M-1, M-2, Induction]
$R(\sigma)$ otherwise [Induction]
$= ((H\delta)/_\sigma\Psi)(\sigma)$
$= (H\delta/\Psi)(\sigma)$

Theorem 3-5

Prove: M_Ψ is a decision mechanism
τ_Ψ is defined as direct and markov.
Below we prove that M_Ψ is homomorphic

1] Assume $\Phi_\Psi(\sigma')$
2] Let $H' \sim H$
3] Let $\tau_\Psi(\sigma') = \sigma$
4] $\tau_\Psi(H'(\sigma')) = (H/\Psi)(\sigma)$ [1,2,3,Lemma M-3]
5] $\tau_\Psi(H'(\sigma')) = (H/_\sigma\Psi)(\sigma)$ [4]
6] $\tau_\Psi(H'(\sigma')) = (\tau_\Psi(H')(\sigma'))(\sigma)$ [1,2,3,5,Lemma M-2]
7] $\tau_\Psi(H'(\sigma')) = (\tau_\Psi(H')(\sigma'))(\tau_\Psi(\sigma'))$ [6,3]
Q.E.D. [1-7]

Theorem 3-6

Given:
1] Ψ is monotonic
2] $\varphi(\sigma) = \Psi(\sigma,\lambda)$

Prove: $\langle\varphi,M_\psi\rangle$ induces Ψ

3] ==> Assume $\Psi(\sigma,H)$
4] $\langle\sigma,H/_\sigma\Psi\rangle \in \{\tau_\psi<\sigma',H'> \mid \varphi_\psi(\sigma')\}$ [Lemma M-2]
5] $\langle\sigma,H\rangle \in \{\tau_\psi<\sigma',H'> \mid \varphi_\psi(\sigma')\}$ [4,1,3, Th 3-9]
6] $\Psi(\sigma,\lambda)$ [3,1]
7] $\varphi(\sigma)$ [6,2]
8] $\langle\sigma,H\rangle \in \{\tau_\psi<\sigma',H'> \mid (M_\psi:\varphi)(\sigma')\}$ [5,7]

9] <== Assume $\langle\sigma,H\rangle \in \{\tau_\psi<\sigma',H'> \mid (M_\psi:\varphi)(\sigma')\}$
10] $\varphi(\sigma)$ [9]
11] $\Psi(\sigma,\lambda)$ [10,2]
12] $\langle\sigma,H\rangle \in \{\tau_\psi<\sigma',H'> \mid \varphi_\psi(\sigma')\}$ [9]
13] $(\exists H_{\not\alpha})(H = H_{\not\alpha}/_\sigma\Psi)$ [12, Lemma M-2]
14] $\Psi(\sigma,H_{\not\alpha}/_\sigma\Psi)$ [11, Th 3-8]
15] $\Psi(\sigma,H)$ [13,14]

Theorem 3-8

Given:
1] $\Psi(\sigma,\lambda)$

Prove: $\Psi(\sigma, H/_\sigma\Psi)$
by induction on length of H
Base $H = \lambda$. Direct by substitution and [1].
Induction Assume for H, prove for Hδ

2] Case 1 $\neg\Psi(\sigma, H/_\sigma\Psi \,\&\, \delta)$
3] $(H\delta)/_\sigma\Psi = H/_\sigma\Psi$ [2]
4] $\Psi(\sigma, (H\delta)/_\sigma\Psi)$ [1,3,Induction]

5] Case 2 $\Psi(\sigma, H/_\sigma\Psi)$
6] $(H\delta)/_\sigma\Psi = H/_\sigma\Psi \,\&\, \delta$ [5]
7] $\Psi(\sigma, (H\delta)/_\sigma\Psi)$ [5,6]

Theorem 3-9

Given:
1] Ψ is monotonic

Prove: $\Psi(\sigma,H) \supset H = H/_\sigma\Psi$
by induction on length of H
Base $H = \lambda$. Direct by substitution.
Induction Assume for H, prove for Hδ

2] Assume $\Psi(\sigma,H\delta)$
3] $\Psi(\sigma,H)$ [2,1]
4] $H = H/_\sigma\Psi$ [3,Induction]
5] $\Psi(\sigma, H/_\sigma\Psi \,\&\, \delta)$ [2,4]
6] $(H\delta)/_\sigma\Psi = H/_\sigma\Psi \,\&\, \delta$ [5]
7] $(H\delta)/_\sigma\Psi = H\delta$ [4,6]

Theorem 3-10

Given:

1] M is strongly consistent
2] M is homomorphic

Prove: M is weakly consistent

3] Assume $\sigma 1' \overset{M}{=} \sigma 2'$
4] $\tau_M(H')(\sigma 1') = \tau_M(H')(\sigma 2')$ [1,3]
5] $\tau_M(\sigma 1') = \tau_M(\sigma 2')$ [3]
6] $(\tau_M(H')(\sigma 1'))(\tau_M(\sigma 1')) = (\tau_M(H')(\sigma 2'))(\tau_M(\sigma 2'))$ [4,5]
7] $\varphi_M(\sigma 1') \wedge \varphi_M(\sigma 2')$ [3]
8] $\tau_M(H'(\sigma 1')) = \tau_M(H'(\sigma 2'))$ [6,7,2]
Q.E.D. [3-8]

Theorem 4-1

Given:

1] φ_{solve} enforces $\varphi_{problem}$
2] $\Psi_{problem}$ is markov
3] $\varphi_{problem}(\sigma) \supset (\forall \delta)(\Psi_{problem}(\sigma, \delta))$

Prove: φ_{solve} enforces $\Psi_{problem}$
That is: $\varphi_{solve}(\sigma) \supset (\forall H)(\Psi_{problem}(\sigma, H))$
by induction on length of H
Base $H = \lambda$. Follows immediately from [2]
Induction Assume for H, prove for Hδ

4] Assume $\varphi_{solve}(\sigma)$
5] $\Psi_{problem}(\sigma, H)$ [4, Induction]
6] $\varphi_{problem}(H(\sigma))$ [1,4]
7] $\Psi_{problem}(H(\sigma), \delta)$ [6,3]
8] $\Psi_{problem}(\sigma, H\delta)$ [5,7,2]

Theorem 4-2

Given:

1] $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\Psi_{problem}$
2] $\langle \varphi_{solve}, M \rangle$ enforces Ψ_{solve}

Prove: $\langle \varphi_{solve}, M \rangle$ enforces $\Psi_{problem}$

3] Assume $(M:\varphi_{solve})(\sigma')$
4] $\varphi_{solve}(\tau_M(\sigma'))$ [3]
5] $(\forall H)(\Psi_{solve}(\tau_M(\sigma'), H) \supset \Psi_{problem}(\tau_M(\sigma'), H))$ [1,4]
6] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>}) \supset \Psi_{problem}(\tau_{M<\sigma', H'>}))$ [5]
7] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>}))$ [2,3]
8] $(\forall H')(\Psi_{problem}(\tau_{M<\sigma', H'>}))$ [6,7]
Q.E.D. [3-9]

Theorem 4-3

Given:

1] $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$
2] $\langle \varphi_{solve}, M \rangle$ enforces Ψ_{solve}
3] M is homomorphic

Prove: $\langle \varphi_{solve}, M \rangle$ enforces $\varphi_{problem}$

4] Assume $(M:\varphi_{solve})(\sigma')$
5] $\varphi_{solve}(\tau_M(\sigma'))$ [4]
6] $(\forall H)(\Psi_{solve}(\tau_M(\sigma'), H) \supset \varphi_{problem}(H(\tau_M(\sigma'))))$ [5,1]
7] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>} \supset \varphi_{problem}((\tau_M(H')(\sigma'))(\tau_M(\sigma'))))$ [6]
8] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>}))$ [2,4]
9] $(\forall H')(\varphi_{problem}((\tau_M(H')(\sigma'))(\tau_M(\sigma'))))$ [7,8]
10] $\varphi_M(\sigma')$ [4]
11] $(\forall H')(\varphi_{problem}(\tau_M(H'(\sigma'))))$ [9,10,3]
Q.E.D. [4-11]

Theorem 4-4

Given:

1] Φ_{solve} is invariant
2] $\Psi_{problem}$ is markov
3] $\Phi_{solve}(\sigma) \supset (\forall \delta)(\Psi_{problem}(\sigma,\delta))$

Prove: Φ_{solve} enforces $\Phi_{problem}$
That is: $\Phi_{solve}(\sigma) \supset (\forall H)(\Psi_{problem}(\sigma,H))$
by induction on length of H
Base $H = \lambda$. Follows immediately from [2]
Induction Assume for H, prove for Hδ

4] Assume $\Phi_{solve}(\sigma)$
5] $\Psi_{problem}(\sigma,H)$ [4, Induction]
6] $\Phi_{solve}(H(\sigma))$ [4,1]
7] $\Psi_{problem}(H(\sigma),\delta)$ [6,3]
8] $\Psi_{problem}(\sigma,H\delta)$ [5,7,2]

Theorem 4-5

Given:

1] $\Phi(\sigma) \equiv \sigma \in \{ \tau_M(\sigma') \mid \Phi'(\sigma') \}$
2] $\Phi_M(\sigma') \equiv \Phi'(\sigma') \vee \neg\Phi(\tau_M(\sigma'))$

Prove: $\Phi' \equiv M:\Phi$

$(M:\Phi)(\sigma')$
$= \Phi_M(\sigma') \wedge \Phi(\tau_M(\sigma'))$
$= (\Phi'(\sigma') \vee \neg\Phi(\tau_M(\sigma'))) \wedge \Phi(\tau_M(\sigma'))$ [2]
$= \Phi'(\sigma') \wedge \Phi(\tau_M(\sigma'))$
$= \Phi'(\sigma')$ [1]

Prove: $\{ \tau_M(\sigma') \mid = \{ \tau_M(\sigma') \mid \Phi_M(\sigma') \}$

$\{ \tau_M(\sigma') \mid \Phi_M(\sigma') \}$
$= \{ \tau_M(\sigma') \mid \Phi'(\sigma') \} \cup \{ \tau_M(\sigma') \mid \neg\Phi(\tau_M(\sigma')) \}$ [2]
$= \{ \tau_M(\sigma') \mid \Phi'(\sigma') \} \cup (\{ \sigma \mid \neg\Phi(\sigma) \} \cap \{ \tau_M(\sigma') \})$
$= (\{ \tau_M(\sigma') \mid \Phi'(\sigma') \} \cup \{ \sigma \mid \neg\Phi(\sigma) \})$
$\quad \cap (\{ \tau_M(\sigma') \mid \Phi'(\sigma') \} \cup \{ \tau_M(\sigma') \})$
$= (\{ \tau_M(\sigma') \mid \Phi'(\sigma') \} \cup \{ \sigma \mid \neg\Phi(\sigma) \}) \cap \{ \tau_M(\sigma') \}$
$= (\{ \sigma \mid \Phi(\sigma) \} \cup \{ \sigma \mid \neg\Phi(\sigma) \}) \cap \{ \tau_M(\sigma') \}$ [1]
$= \{ \tau_M(\sigma') \}$

Theorem 5-2

Given:

1] $\varphi_{solve} \subseteq \varphi_{problem}$
2] φ_{solve} is Ψ_{solve}-invariant

Prove: $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$

3] Assume $\varphi_{solve}(\sigma)$
4] Assume $\Psi_{solve}(\sigma, H)$
5] $\varphi_{solve}(H(\sigma))$ [2,4,3]
6] $\varphi_{problem}(H(\sigma))$ [5,1]
Q.E.D. [(3,4)-6]

Theorem 5-3

Given:

1] Ψ is markov
2] $\Psi(\sigma, \delta) \supset . \varphi(\sigma) \supset \varphi(\delta(\sigma))$

Prove: φ is Ψ-invariant
That is: $\Psi(\sigma, H) \supset . \varphi(\sigma) \supset \varphi(H(\sigma))$
by induction on length of H
Base $H = \lambda$. Direct by Substitution.
Induction Assume for H, prove for $H\delta$

3] Assume $\Psi(\sigma, H\delta)$
4] $\Psi(\sigma, H)$ [3,1]
5] $\varphi(\sigma) \supset \varphi(H(\sigma))$ [4, Induction]
6] $\Psi(H(\sigma), \delta)$ [3,1]
7] $\varphi(H(\sigma)) \supset \varphi((H\delta)(\sigma))$ [6,2]
8] $\varphi(\sigma) \supset \varphi((H\delta)(\sigma))$ [5,7]
Q.E.D. [3-8]

Theorem 6-1

Given:

1] $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\varphi_{problem}$
2] $\langle \varphi_{solve}, M \rangle$ produces Ψ_{solve}
3] M is homomorphic

Prove: $\langle \varphi_{solve}, M \rangle$ produces $\varphi_{problem}$

4] Assume $(M:\varphi_{solve})(\sigma')$
5] $\varphi_{solve}(\tau_M(\sigma'))$ [4]
6] $(\forall H)(\Psi_{solve}(\tau_M(\sigma'), H) \supset \varphi_{problem}(H(\tau_M(\sigma'))))$ [5,1]
7] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>}) \supset \varphi_{problem}((\tau_M(H'))(\sigma'))(\tau_M(\sigma'))))$ [6]
8] $(\exists H')(\Psi_{solve}(\tau_{M<\sigma', H'>}))$ [2,4]
9] $(\exists H')(\varphi_{problem}((\tau_M(H'))(\sigma'))(\tau_M(\sigma'))))$ [7,8]
10] $\varphi_M(\sigma')$ [4]
11] $(\exists H')(\varphi_{problem}(\tau_M(H'(\sigma'))))$ [9,10,3]
Q.E.D. [4-11]

Theorem 6-2

Given:

1] $\langle \varphi_{solve}, \Psi_{solve} \rangle$ enforces $\Psi_{problem}$
2] $\langle \varphi_{solve}, M \rangle$ produces Ψ_{solve}

Prove: $\langle \varphi_{solve}, M \rangle$ produces $\Psi_{problem}$

3] Assume $(M:\varphi_{solve})(\sigma')$
4] $\varphi_{solve}(\tau_M(\sigma'))$ [3]
5] $(\forall H)(\Psi_{solve}(\tau_M(\sigma'), H) \supset \Psi_{problem}(\tau_M(\sigma'), H))$ [1,4]
6] $(\forall H')(\Psi_{solve}(\tau_{M<\sigma', H'>}) \supset \Psi_{problem}(\tau_{M<\sigma', H'>}))$ [5]
7] $(\exists H')(\Psi_{solve}(\tau_{M<\sigma', H'>}))$ [2,3]
8] $(\exists H')(\Psi_{problem}(\tau_{M<\sigma', H'>}))$ [6,7]
Q.E.D. [3-8]

Theorem 7-4

Given:

1] $X(\Phi,M)$ ≡ $\langle\Phi,M\rangle$ enforces $\Psi_{problem}$

Prove: $\langle\Phi2,M2\rangle \subseteq \langle\Phi1,M1\rangle \supset X(\Phi1,M1) \supset X(\Phi2,M2)$

2] Assume $\langle\Phi2,M2\rangle \subseteq \langle\Phi1,M1\rangle$

3] Assume $X(\Phi,M1)$

4] $\{ \tau_{M2}\langle\sigma',H'\rangle \mid (M2:\Phi2)(\sigma') \} \subseteq \{ \tau_{M1}\langle\sigma',H'\rangle \mid (M1:\Phi1)(\sigma') \}$ [2]

5] $\langle\sigma,H\rangle \in \{ \tau_{M1}\langle\sigma',H'\rangle \mid (M1:\Phi1)(\sigma') \} \supset \Psi_{problem}(\sigma,H)$ [1,3]

6] $\langle\sigma',H\rangle \in \{ \tau_{M2}\langle\sigma',H'\rangle \mid (M2:\Phi2)(\sigma') \} \supset \Psi_{problem}(\sigma,H)$ [4,5]

7] $X(\Phi2,M2)$ [1,6]

Q.E.D. [(2,3)-7]

Prove: $X(\Phi1,M) \wedge X(\Phi2,M) \supset X(\Phi1 \vee \Phi2, M)$

proof follows immediately from proof of the following lemma

$M:\Phi1 \vee M:\Phi2 \equiv M:(\Phi1 \vee \Phi2)$

$(M:\Phi1)(\sigma') \vee (M:\Phi2)(\sigma')$

= $\{ \Phi_M(\sigma') \wedge \Phi1(\tau_M(\sigma')) \} \vee \{ \Phi_M(\sigma') \wedge \Phi2(\tau_M(\sigma')) \}$

= $\Phi_M(\sigma') \wedge \{ \Phi1(\tau_M(\sigma')) \vee \Phi2(\tau_M(\sigma')) \}$

= $(M:(\Phi1 \vee \Phi2))(\sigma')$

Theorem 7-5

Given:

1] $X(\Phi1) \wedge X(\Phi2) \supset X(\Phi1 \vee \Phi2)$

2] Φ_{max} maximally solves X

Prove: $\Phi_{max} \equiv \vee \{ \Phi \mid X(\Phi) \}$

3] Let $\Phi_{join} \equiv \vee \{ \Phi \mid X(\Phi) \}$

4] $X(\Phi_{max})$ [2]

5] $\Phi_{max} \subseteq \Phi_{join}$ [3,4]

6] $X(\Phi_{join})$ [1,3]

7] $\Phi_{max} \equiv \Phi_{join}$ [5,6,2]

Q.E.D. [3,7]

Theorem 7-6

Given:

1] $X(\Phi\star,M\star)$ ≡ $\langle\Phi\star,M\star\rangle$ enforces $\Psi_{problem}$

2] *<Worth,≤> is a monotonic measure*

3] $\langle\Phi,M\rangle$ induces $\Psi_{problem}$

Prove: $\langle\Phi,M\rangle$ optimally solves X

4] $X(\Phi\star,M\star) \supset \langle\Phi\star,M\star\rangle \subseteq \langle\Phi,M\rangle$ [1,3, Th 7-1]

5] $X(\Phi,M)$ [1,3]

6] $Worth(\Phi,M) = Max \{ Worth(\Phi\star,M\star) \mid X(\delta,M\star) \}$ [2,4,5]

Theorem 7-7

Given:

1] <Worth,≤> is a monotonic measure

2] Φ_{max} maximally solves X

3] $X(\Phi1) \wedge X(\Phi2) \supset X(\Phi1 \vee \Phi2)$

Prove: Φ_{max} optimally solves X

4] $\Phi_{max} \equiv \vee \{ \Phi \mid X(\Phi) \}$ [2,1, Th 7-5]

5] $Worth(\Phi_{max}) = Worth(\vee \{ \Phi \mid X(\Phi) \})$ [4]

6] $Worth(\Phi_{max}) = Max \{ Worth(\Phi) \mid X(\Phi) \}$ [5,1]

Appendix C - References

[Atwood 72] J Atwood, ed. "Project Sue as a Learning Experience", U Toronto, CSRG-19, Sept 1972

[Belpaire 75] G Belpaire, "Synchronization: Is a synthesis of problems possible?", ACM Sigcomm-Sigplan Interprocess Communication Workshop, March 1975

[Campbell & Habermann 75] R Campbell, N. Habermann, "The Specification of Process Synchronization by Path Expressions", Proceedings International Symposium on Operating Systems Theory and Practice, April 1974

[Cohen 75] E Cohen, "Semantic Models for Parallel Systems", CMU TR Jan 1975

[Cohen 76] E Cohen, "Strong Dependency: A Formalism for Describing Information Transmission in Computational Systems", CMU TR August 1976

[Cohen & Jefferson 75] E Cohen, D Jefferson, "Protection in the HYDRA Operating System", Proceedings 5th Symposium on Operating System Principles, Nov 1975 (also SIGOPS, v9, 5)

[Cosserat 74] D Cosserat, "A Data Model based on the Capability Protection Mechanism", International Workshop on Protection in Operating Systems, IRIA 1974

[Courtois, Heymans & Parnas 72] P Courtois, F Heymans, D Parnas, "Concurrent Control with 'Readers' and 'Writers'", CACM v14,10 (Oct 1971)

[Crisman 65] P Crisman, ed. "The Compatible Time-Sharing System: A Programmer's Guide", MIT Press 1965

[Denning 75] D Denning, "Secure Information Flow in Computer Systems", PhD Thesis, Comp Sci Dept, Purdue Univ, May 1975

[Denning 76] D Denning, "A Lattice Model of Secure Information Flow", CACM v19,5 (May 1976)

[Dennis & Van Horn 66] J Dennis, E Van Horn, "Programming Semantics for Multiprogrammed Computation", CACM v17,7 (March 1966)

[Dijkstra 68a] E Dijkstra, "Cooperating Sequential Processes", In Programming Languages, F Genuys, ed., Academic Press 1968

[Dijkstra 68b] E Dijkstra, "The Structure of the 'THE'-Multiprogramming System", CACM v11,5 (May 1968)

[Dijkstra 72] E Dijkstra, "A Class of Allocation Strategies Inducing Bounded Delay only", Spring Joint Computer Conference, 1972

[Graham & Denning 72] R Graham, P Denning, "Protection - Principles and Practice", Spring Joint Computer Conference, 1972

[Gray 72] J Gray, B Lampson, B Lindsay, H Sturgis, "The Control Structure of an Operating System", IBM Research, RC 3949, July 1972

[Gray 75] J Gray, R Lorie, G Putzolu, I Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", IBM Research, RJ 1654, Sept 1975

[Greif 75] I Greif, "Semantics of Communicating Parallel Processes", PhD Thesis, MIT MAC-TR-154, Sept 1975

[Greif & Hewitt 75] I Greif, C Hewitt, "Actor Semantics of PLANNER-73", 2nd ACM Symp on Principles of Programming Languages, Jan 1975

[Griffiths 74] P Griffiths, "Synver: A System for the Automatic Synthesis and Verification of Synchronization Processes", Harvard Center for Research in Computing Technology, TR 22-74, 1974

[Hansen 73] P Brinch Hansen, "Operating System Principles", Prentice-Hall 1973

[Harrison, Ruzzo, Ullman 75] M Harrison, M Ruzzo, J Ullman, "On Protection in Operating Systems", 5th Symp. on Operating System Principles, Nov 1975

[Hewitt 74] C Hewitt, "Protection and Synchronization in Actor Systems", Working Paper 83, AI Lab MIT, Nov 1974

[Hoare 74] C A R Hoare, "Monitors: An OPerating System Structuring Concept", CACM v17,10 (Oct 1974)

[Hopcroft & Ullman 69] J Hopcroft, J Ullman, "Formal Languages and their Relation to Automata", Addison Wesley, 1969

[Jones 73] A Jones, "Protection in Programmed Systems", CMU PhD Thesis, June 1973

[Jones & Lipton 75] A Jones, R Lipton, "The Enforcement of Security Policies for Computations", 5th Symp. on Operating System Principles, Nov 1975

[Jones & Wulf 74] A Jones, W Wulf, "Towards the Design of Secure Systems", International Workshop on Protection in Operating Systems, IRIA, 1974

[Lampson 71] B Lampson, "Protection", 5th Annual Princeton Conference on Information Sciences and Systems, March 1971

[Lampson 73] B Lampson, "A Note on the Confinement Problem", CACM v16,10 (Oct 1973)

[Levin 75] R Levin, E Cohen, W Corwin, F Pollack, W Wulf, "Policy/Mechanism Separation in Hydra", 5th Symp. on OPerating System Principles, Nov 1975

[Lipton 73] R Lipton, "On Synchronization Primitive Systems", Yale CSRR, 22, 1973 (also CMU PhD Thesis)

[Lipton 75] R Lipton, "Reduction: A Method of Proving Properties of Parallel Programs", CACM v18,12 (Dec 1975)

[Millen 76] J Millen, "Security Kernal Validation in Practice", CACM v19,5 (May 1976)

[Milner 72] R Milner, "Processes: A Mathematical Model of Computing Agents", Proc. Logic Colloquium, Bristol 1972

[Needham 72] R Needham, "Protection Systems and Protection Implementations", FJCC 1972

[Organick 72] E Organick, "The MULTICS System: An Examination of its Structure", MIT Press 1972

[Parnas 72] D Parnas, "A Technique for Software Module Specification with Examples", CACM v15,5 (May 1972)

[Patil 71] S Patil, "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes", MIT MAC-CSG-57, Feb 1971

[Peuto 74] B Peuto, "A Comparative Study of Real Estate Law and Protection Systems", PhD Thesis, UC Berkeley, ERL-M439, May 1974

[Popek & Goldberg 74] G Popek, R Goldberg, "Formal Requirements for Virtualizable Third Generation Arvhitectures", CACM v17,7 (July 1974) [Price 73] W Price, "Implications of a Virtual Machine Mechanism for Implementing Protection in a Family of Operating Systems" PhD Thesis, CMU, June 1973

[Redell 74] D Redell, "Naming and Protection in Extendible Operating Systems", PhD Thesis, UC Berkeley. Also MIT MAC-TR-140 Nov 1974

[Redell & Fabry 74] D Redell, B Fabry, "Selective Revocation of Capabilities", International Workshop on Protection in Operating Systems, IRIA, 1974

[Riddle 73] W Riddle, "A Method for the Description and Analysis of Complex Systems", Sigplan Notices v8,9 (Sept 1973)

[Robinson & Holt 74] L Robinson, R Holt, "Formal Specifications for Solutions to Synchronization Problems", SRI, 1974

[Robinson 75] L Robinson etal, "A Formal Methodology for the Design of Operating System Software", *Proc 1975 Conference on Reliable Software*, Apr 1975

[Rotenberg 73] L Rotenberg, "Making Computers Keep Secrets", PhD Thesis MIT, MAC-TR-116, Sept 1973

[Saltzer 74] J Saltzer, "Protection and the Control of Information Sharing in MULTICS", CACM v17,7 (July 1974)

[Schneider 76] E Scneider, "Synchronization of Finite State Shared Resources", CMU PhD Thesis, March 1976

[Schroeder 72] M Schroeder, "Cooperation of Mutually Suspicious Subsystems", PhD Thesis, MIT, MAC-TR-104, Sept 1972

[Schroeder & Saltzer 72] M Schroeder, J Saltzer, "A Hardware Architecture for Implementing Protection Rings", CACM v15,3 (March 1972)

[Shannon & Weaver 49] C Shannon, W Weaver, "The Mathematical Theory of Communication", U Illinois Press, 1949

[Vantilborgh & Lamsweerde 72] H Vantilborgh, A van Lamsweerde, "On an Extension of Dijkstra's Semaphore Primitives", Information Processing Letters, v1, 5, Oct 1972

[Weissman 69] Weissman C, "Security Controls in the Adept-50 Time-Sharing System" FJCC 1969

[Wodon 72] P Wodon, "Still another Tool for Controlling Cooperating Algorithms", CMU-TR 1972

[Wulf 74] W Wulf, et.al., "HYDRA: The Kernel of a Multiprocess Operating System", CACM v17,6 (June 1974)